POLITECNICO DI TORINO

Master's Degree in Embedded Computing Systems

Master's Degree Thesis

Modeling and Design of SRAM-Based Physical Unclonable Function

Supervisor

Prof. Massimo PONCINO

Candidate

Erik SIVERTSEN

October 2024

Abstract

With the proliferation of IoT and other distributed systems, securing embedded devices against physical attacks has become increasingly important. Conventional techniques such as storing secret keys in non-volatile memory have been shown to be vulnerable to invasive attacks, highlighting the need for new methods of protecting such systems and reliably authenticating devices.

This thesis explores Physical Unclonable Functions (PUFs) as a security primitive to ensure that secret keys and identifiers are not available to potential attackers. Specifically, the focus is on SRAM PUFs, which use memory start-up values to generate unique IDs for manufactured devices. SRAM is present on virtually all modern embedded systems, and its volatile nature means that keys are not stored in the device while it is powered down but can be recreated each time it is powered on.

To qualify for use in security-critical applications, the produced keys must be truly random and unique to each device. Therefore, a set of metrics to quantify the strength of the keys is presented and used to evaluate a data set extracted from a custom test chip. The experiments compare a reference standard SRAM against a memory that has been modified to allow fast erasure and fast initialization by destabilizing the cells. This modification enables rapidly performing multiple evaluations of the memory to reduce the noise present in the PUF response.

Results show that the modifications have some negative impact on the security metrics of the resulting keys, meaning it is advisable to implement privacy amplification techniques before using the PUF in security applications.

Acknowledgements

Thank you to the staff at CEA Grenoble for the opportunity to conduct the research for this thesis at their laboratories and benefit from their expertise. I especially want to thank María Ramírez Corrales and Lorenzo Ciampolini at CEA and Professor Massimo Poncino at PoliTO for supervising the thesis. Lastly, I want to thank the EMECS programme for providing an excellent educational and personal experience over the past two years.

Table of Contents

List of Tables

List of Figures

Glossary

AES	Advanced Encryption Standard
ASIC	Application-Specific Integrated Circuit
BER	Bit Error Rate
C-SRAM	Computational-SRAM
CRP	Challenge-Response Pair
DAA	Directed Accelerated Aging
ECC	Error Correcting Code
FDSOI	Fully Depleted Silicon-On-Insulator
FEFI	Fast Erase Fast Initialization
fHD	Fractional Hamming Distance
FPGA	Field Programmable Gate Array
HMAC	Hash-based Message Authentication Code
HW	Hamming Weight
IoT	Internet of Things
IP	Intellectual Property
ISA	Instruction Set Architecture
LTK	Lines To Key

MCC	Memory Cell Classification
MOS	Metal Oxide Semiconductor
MSB	Most Significant Bit
NBTI	Negative Bias Temperature Instability
NMC	Near-Memory Computing
NVM	Non-Volatile Memory
PUF	Physical Unclonable Function
RDF	Random Dopant Fluctuations
RNG	Random Number Generator
RTL	Register-Transfer Level
SIMD	Single Instruction Multiple Data
SRAM	Static Random-Access Memory
TMV	Temporal Majority Voting
WSN	Wireless Sensor Network

Chapter 1

Introduction

Modern embedded devices often run applications requiring secure authentication to ensure system integrity. Achieving this is increasingly complicated by the proliferation of embedded systems into new applications in which devices must operate autonomously in areas where potential attackers could gain physical access to the device. Since user input such as passwords can not be relied upon as a root of trust in these applications, the current solution to this problem is to store secret keys in non-volatile memory (NVM) [1]. This method is, however, subject to invasive physical attacks where an attacker may extract the secret keys through techniques such as board-level probing [2]. Protecting against this kind of attack requires additional circuitry to detect and prevent tampering, which is often expensive in terms of both cost and area and relies on the system being continuously powered, leaving the potential attack vector of powering down the device. Storing the key in volatile memory instead would mitigate this vulnerability but would significantly hamper the autonomous operation of the device by requiring manual intervention to re-provision the key(s) if power is lost temporarily.

To address these shortcomings of current methods, novel technologies allowing secure authentication, autonomous operation, and physical integrity are needed. One promising contender is the Physical Unclonable Function (PUF), based on the naturally occurring variations in the semiconductor manufacturing process [1]. Minor differences between manufactured dies can provide a unique *fingerprint* for each device resulting from random variations, making the identifier unclonable. PUF keys are only present in the device while powered, making them harder to extract by invasive attacks.

Since PUFs rely on variance in the physical characteristics of a die, they are also subject to noise from environmental factors such as temperature, voltage, and age. To overcome this issue, most current implementations of PUF rely on error correction. This is done by programming error-correcting codes into each device before deployment, in a stage known as enrollment. This enrollment phase introduces costs and potential security issues into the deployment process for embedded devices [3].

This thesis is developed as part of an internship at CEA Grenoble. CEA (*Commissariat à l'énergie atomique et aux énergies alternatives*), or the French Alternative Energies and Atomic Energy Commission, is a public research institution for fundamental and technology research headquartered in Paris with sites across France. At its Innovative Functions for Mixed Circuits Laboratory (LFIM) in Grenoble, it researches novel uses for integrated circuits, including PUFs.

A prototype PUF implementation has previously been developed at CEA, and this thesis aims to characterize and improve upon this implementation. Additionally, we will investigate the feasibility of error correction-free PUFs by using post-processing techniques to reduce or eliminate noise in PUF responses, removing the need for an enrollment phase. This would reduce the time required to deploy embedded systems containing PUFs and potentially increase security as there is no need to store error correction helper data in non-volatile memory. The thesis findings will inform the future development of a PUF IP block.

Chapter 2 contains the required background information on PUFs and introduces a system of binary notations, which will be used throughout this thesis. Lastly, it defines the quality metrics by which PUFs are evaluated. Next, chapter 3 presents previous works on the subject, which will guide the development work of this thesis, including technologies that will be core to the design of PUF key extraction functions.

Chapter 4 describes the tools used in the development work and the experiments conducted, whose results are presented in chapter 5. Finally, conclusions and future recommendations are presented in chapter 6.

Chapter 2

Background

Authenticating participants in communications or transactions is a daily occurrence in the modern world, from inputting passwords or biometrics to showing identity documents. This also extends into computer networks, where authentication typically occurs using cryptographic keys, which must be kept secret from potential attackers. In distributed systems such as IoT or Wireless Sensor Networks (WSN), this poses a significant challenge since attackers could gain physical access to the devices, meaning secret keys have to be protected from invasive attacks [4].

With full access to a device, many attacks could be performed to extract secret information, such as microprobing or fault injection [2]. Defending against these attacks can be difficult and costly, inspiring the need for novel security primitives that enable secure authentication in insecure environments.

This chapter presents the Physically Unclonable Function (PUF), including its history and construction. To evaluate the qualities of a PUF, we need to define some metrics by which it can be assessed. In section 2.3, five metrics for analyzing PUF quality are defined, but before that, a system of binary notations will be introduced, which will be used for the rest of this thesis.

2.1 Physical Unclonable Functions

Physical unclonable functions (PUF), first introduced by R. S. Pappu [5], are functions that are based upon the unique physical properties of an object and can not be reproduced systematically, except by the object itself. Unlike normal deterministic functions, where the result is expected to be the same no matter where the function is evaluated, each PUF is expected to have its unique response. Unlike a truly random function, a PUF is expected to give the same response each time it is evaluated. PUFs are unclonable because the function's parameters are not determined by design but by natural variations. Therefore, reliably predicting or reproducing the function should not be possible, even with complete knowledge of its design and manufacturing method.

The initially proposed architecture to achieve such a function was to shine a laser through an epoxy wafer implanted with silicon spheres and measure the resulting light scattering to obtain a unique object fingerprint [5]. Gassend et al. [6] later introduced the silicon-based PUF using the naturally occurring process variations in semiconductor manufacturing, such as random dopant fluctuations (RDF) or variance in capacitance and signal propagation delay. For silicon PUFs, the original proposal was to use signal delay variations in a self-oscillating loop circuit to produce a unique device signature [6]. The SRAM-based PUF was introduced by Guajardo et al. [7] in 2007 and is the focus of this project.

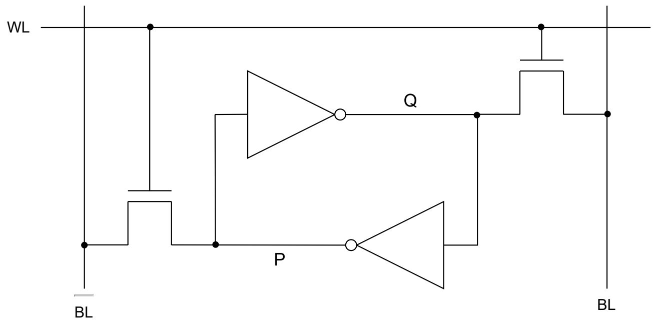


Figure 2.1: 6T-SRAM cell

Figure 2.1 shows a standard 6T-SRAM cell made from two cross-coupled inverters forming a bi-stable circuit. If this circuit is ideally perfectly balanced, the initialization value of the cell is a truly random outcome at each new circuit power-on. However, because of RDFs introduced at manufacturing time, the threshold voltage of the transistors will fluctuate from one cell to another, making each cell biased towards one state. This characteristic makes SRAM, present on virtually all modern embedded systems, feasible as the basis for a PUF.

To be useful for authentication purposes, PUFs have to be time-invariant and provide sufficiently random patterns to represent a unique fingerprint for a given object. Literature on the subject distinguishes between so-called *strong* and *weak* PUFs, differentiated by the number of *Challenge-Response Pairs* (CRP) that they can support. These CRPs are part of a protocol where an agent seeking to authenticate a device issues a *challenge*, which is then used by the device to generate a response from its unique physical properties [1]. Weak PUFs can generate only a few or even just a single such response, whereas strong PUFs can support many CRPs. The full PUF protocol consists of an enrollment phase before deployment and an authentication stage during use.

Conventionally, for strong PUFs, the devices undergo an enrollment phase where a large set of CRPs are recorded. During operation, the CRPs are consumed as they are used since an attacker could record the responses [1]. Weak PUFs are conventionally used together with cryptographic functions such as Hash-based Message Authentication Code (HMAC) where the response is kept secret since they only support a small number of CRPs [1].

In addition to using the PUF response directly to authenticate a die, other applications have also been proposed. The PUF can be used as a root of trust, where provisioned keys are wrapped using the PUF root key to be stored safely in NVM [8]. The keys are then intrinsically tied to the device, which can unwrap and utilize them. In their original paper on silicon PUFs, Gassend et al. [6] describe the potential to use PUFs for software licensing, tying code to a specific device to mitigate software piracy. Guajardo et al. [7] similarly describe using PUFs for IP protection in FPGAs by encrypting the bit-stream to prevent cloning. Since most PUFs are inherently noisy, they can also be used as true random number generators by isolating the noise they produce [9].

2.2 Binary Notations

In the case of digital circuits and algorithms, one uses *words* that are built using *bits*, i.e., symbols taken from the binary alphabet 0, 1.

The following text refers to the notation developed in [10], where PUF keys of length L are measured S times on D different devices. For each dimension (key length, number of samples, number of devices) the correspondent index is the lowercase index l, s or d.

For a word B of length L, we denote as b_l the value $\in \{0, 1\}$ of the l-th bit, with $1 \leq l \leq L$. An SRAM word being repeatedly measured S times will have thus S different *samples* of the same word, denoting as $b_{s,l}$ the value $\in \{0, 1\}$ of the l-th bit, with $1 \leq l \leq L$, at the s-th measurement, with $1 \leq s \leq S$. Finally, we will denote with $b_{s,l,d}$ the value $\in \{0, 1\}$ of the l-th bit, with $1 \leq l \leq L$, at the s-th measurement, with $1 \leq s \leq S$, on d-th device, with $1 \leq d \leq D$.

Indexes will be dropped when not significant. For instance, the number of bits being equal to one in a word B is given by

$$n_1(B) = \sum_{l=1}^{L} b_l = L \times HW(B) \tag{2.1}$$

where the last equation defines the *Hamming Weight* of a word.

In general, given two words B and C having common length L, one indicates the number of bits by which they differ as *Hamming Distance* or $HD(B,\ C)$:

$$HD(B,\ C) = \sum_{l=1}^{L} b_l \oplus c_l = L \times fHD(B,\ C) \tag{2.2}$$

where the last equation defines the *fractional HD* between two words and the $\oplus$ operator indicates the XOR logical operation.

Indicating the null word composed of 0s only as $\emptyset$, one has combining 2.1 and 2.2:

$$L \times HW(B) = \sum_{l=1}^{L} b_l = \sum_{l=1}^{L} b_l \oplus 0 = HD(B,\ \emptyset) = L \times fHD(B,\emptyset) \tag{2.3}$$

In the most general way, one can write a word measurement by developing its bits:

$$B_{s,d} = b_{s,1,d}\ b_{s,2,d} \cdots b_{s,L,d} \tag{2.4}$$

Once S measurements have been done, the best possible estimation for the probability of each bit to initialize to 1 is

$$\beta_{l,d} = \frac{1}{S}\sum_{s=1}^{S} b_{s,l,d} \quad \beta_{l,d} \in [0,\ 1] \tag{2.5}$$

Once S measurements have been carried out on word B_d, one can apply *Temporal Majority Voting* to determine which is the *golden value* of this word, simply finding for each bit which is its most probable value $\hat{b}_{l,d} \in \{0,\ 1\}$, depending on which value occurred with the highest frequency. With the current notation, it is straightforward to obtain the following relationship:

$$\hat{b}_{l,d} = floor(0.5 + \beta_{l,d}) = \begin{cases} 0 \ if \ \beta_{l,d} < 0.5 \\ 1 \ if \ \beta_{l,d} \geq 0.5 \end{cases} \tag{2.6}$$

Using this notation, the golden (or TMV) word for device d will be indicated as $\hat{b}_d$ and its l-th bit as $\hat{b}_{l,d}$.

It is important to assess the average properties of a TMV word. For this purpose, we might calculate its average value

$$\bar{\hat{b}}_d = \frac{1}{L}\sum_{l=1}^{L} \hat{b}_{l,d} \tag{2.7}$$

while the average over different words (devices) of a single bit will be expressed with bra and ket parenthesis:

$$< \hat{b}_l >= \frac{1}{D} \sum_{d=1}^{D} \hat{b}_{l,d} \tag{2.8}$$

Combining 2.7 and 2.8 yields the so-called *bit uniformity* $Unif$:

$$Unif =< \bar{\hat{b}} >= \frac{1}{L} \sum_{l=1}^{L} < \hat{b}_l >= \frac{1}{D} \sum_{d=1}^{D} \bar{\hat{b}}_d \tag{2.9}$$

2.3 Quality Metrics

To evaluate the results of experiments and make them comparable to other research on PUFs, we require a standardized set of metrics. In this section, the metrics *Uniformity, Aliasing, Auto-correlation, Uniqueness* and *Reliability* are presented, which were formalized by researchers at Virginia Polytechnic Institute and State University (Virginia Tech) [11].

2.3.1 Uniformity

The number of possible combinations of a set of symbols depends on the proportions of each symbol in the word. In the case of binary words, this means the proportion of ones to zeros. Suppose the device systematically produces only ones or only zeroes. In that case, there is just one possible combination, while the maximum number of possible combinations is attained when the distribution is even. If one could learn that a class of devices systematically produces a higher proportion of any symbol, this would reduce the security of the identifier as there are fewer possible combinations that the key could take. Hamming Weight (HW), which from equation 2.3 can be defined as the fHD to the all-zero word, can be used to quantify this property and should ideally be close to $\frac{1}{2}$.

$$HW(\hat{B}_d) = fHD(\hat{B}_d, \emptyset) = \frac{1}{L} \sum_{l=1}^{L} \hat{b}_{l,d} \qquad \forall d \in [1..D] \tag{2.10}$$

2.3.2 Aliasing

While *uniformity* examines whether the PUF response is biased towards any symbol, it does not reveal whether any of the individual bits in the response is biased, meaning the bit at some position often or always takes a particular value. To examine this property, we define *aliasing* as the Hamming Weight of the golden value in each bit position l across all devices. If there is no correlation between devices, then the *aliasing* in each bit position should be near $\frac{1}{2}$.

$$aliasing(l) = <\hat{b}_l> = \frac{1}{D}\sum_{d=1}^{D}\hat{b}_{l,d} \qquad \forall l \in [1..L] \tag{2.11}$$

2.3.3 Auto-correlation

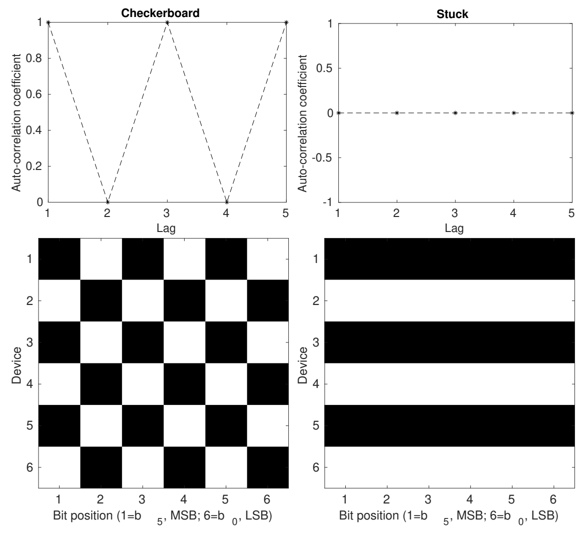


Figure 2.2: Auto-correlation of example patterns

Even together, *uniformity* and *aliasing* cannot detect recurring patterns in the PUF responses where bits in the response are related. To detect this, we define *auto-correlation* as the average relation between bits at a distance/lag j. If there is no correlation, this value should be close to $\frac{1}{2}$. A value close to zero implies a positive correlation, and a value close to one indicates a negative correlation.

$$auto\text{-}correlation(j) = \frac{1}{D(L-j)}\sum_{d=1}^{D}\sum_{l=j+1}^{L}\hat{b}_{l,d} \oplus \hat{b}_{l-j,d} \qquad \forall j \in [1..L-1] \tag{2.12}$$

Figure 2.2 shows this metric applied to two example patterns. In the checkerboard example, the auto-correlation coefficient at each odd number is one, showing negative correlation, since each bit is the opposite of all bits at an odd distance. Similarly, each coefficient at an even distance is zero, showing positive correlation, since all bits at an even distance are the same. In the stuck example where each key contains only one symbol, all coefficients are zero since all bits are positively correlated.

2.3.4 Uniqueness (inter-PUF fHD)

The identifier generated by a PUF must be unpredictable and uniquely tied to a specific device, meaning it must avoid systematic bias toward taking certain values. If devices of the same design and manufacture produce a similar signature, then the security of the device identifier is significantly reduced, as the knowledge of a single device can be used to extrapolate the signature of other devices of the same class. The fractional Hamming Distance (fHD) is used to evaluate this property, which measures the proportion of bits that differ between binary words of length L. If the signatures $\hat{B}$ produced by two devices $d1$ and $d2$ of the same class are indeed uncorrelated, then the fHD between them should be close to $\frac{1}{2}$. This metric is referred to as the inter-class or inter-PUF fHD.

$$fHD(\hat{B}_{d1}, \hat{B}_{d2}) = \frac{1}{L} HD(\hat{B}_{d1}, \hat{B}_{d2}) = \frac{1}{L} \sum_{l=1}^{L} \hat{b}_{l,d1} \oplus \hat{b}_{l,d2} \tag{2.13}$$

2.3.5 Reliability (Intra-PUF fHD)

For a PUF to function as a device signature, it has to be reliable over time, meaning we can extract the same output from the device at each evaluation. Reliability for embedded devices means reproducing the same output under all operating conditions in which the device is expected to function. Virtually no PUFs are 100% reliable over time, which means tolerating some error between evaluations is necessary. Since cryptographic applications require the same input each time to produce the same output, this tolerance usually implies some form of error correction.

Error Correcting Codes (ECC) such as Hamming Codes are well studied and already widely used in modern integrated circuits [12]. These methods rely on storing some additional helper data to recreate the target output from an erroneous input, which in the case of PUFs would imply storing this data in NVM for recreating the key. Storing this helper data in NVM negatively impacts security, as it can give away some information about the key itself. The size of the needed helper data depends on how many bits it is necessary to correct, meaning that a more reliable PUF requires less helper data, which in turn leaks less information about the key.

The metric best suited to characterize reliability depends on the application. For some applications, the average fHD between responses $s1$ and $s2$ from the same device d, called the intra-class or intra-PUF fHD, is used. This measure should ideally be as close to zero as possible. In other applications, reliability is measured as how often the PUF produces an expected/golden response. We will use the distance to the expected/golden response $\hat{B}$ for our purposes.

$$fHD(B_{s,d},\hat{B}_d) = \frac{1}{L}HD(B_{s,d},\hat{B}_d) = \frac{1}{L}\sum_{l=1}^{L} b_{l,s,d} \oplus \hat{b}_{l,d} \qquad \forall\{s,d\} \tag{2.14}$$

Some publications also refer to this metric as the Bit Error Rate (BER), although this term is more commonly used in digital transmission.

We will also refer to the proportion of times the PUF produces the expected/-golden response with no errors, calculated as the complement of the proportion of responses with an error.

$$p_{zero\ err} = 1 - \frac{1}{D \times S}\sum_{d=1}^{D}\sum_{s=1}^{S}\lceil fHD(B_{s,d},\hat{B}_d)\rceil \tag{2.15}$$

Chapter 3

Related Work

This chapter will introduce scientific works on Physical Unclonable Functions, informing the research and development process described later in this thesis. In addition to the academic works that will be presented, it is also worth highlighting that PUF technology has already been successfully commercialized and is available in several products that will be presented.

3.1 Environmental Effects

Since being proposed as a basis for a PUF, SRAM spontaneous initialization has been extensively studied, with results overwhelmingly confirming that SRAM initialization values can produce unique and reliable identifiers. While the raw SRAM initialization values have been found to be highly correlated over time, significant noise is also present between initializations. Several factors have been found to contribute to this noise:

3.1.1 Manufacturing

The process used to manufacture the SRAM has been shown to impact the quality of the PUF, with researchers at Broadcom [13] finding that using a FinFET process yields better randomness than a planar process. Clark et al. [14] also find in a study that the process corner impacts PUF operation. The authors demonstrate that the slow-PMOS slow-NMOS (SS) process corner exhibits the lowest intra-PUF fHD and has the most stable bits, followed by the typical-PMOS typical-NMOS (TT) corner and the fast-PMOS fast-NMOS (FF) process corner having the worst PUF performance. Lastly, Holcomb et al. [9] find that standalone SRAM chips are more reliable as PUFs than embedded SRAMs.

The material used in the manufacturing could also impact the noise levels in the PUF. O'uchi et al. [15] demonstrate in a paper that SRAM cells manufactured using polysilicon as channel material are, on average, more stable than those made using crystal silicon, showing an improvement in inter-PUF fHD by a factor of 3.4. The poly-Si devices are shown to have approximately three times the number of completely stable bits compared to the crystal-Si devices.

3.1.2 Voltage & Temperature

During deployment, the SRAM operation is also affected by its operating conditions: In a study by Selimis et al. [4] on the effect of operating conditions on SRAM, the authors find that while the final operating voltage of the circuit has little impact on noise levels, the voltage ramp up time has a significant effect on the SRAM initialization. These results are confirmed in a study published by the University of Naples [16] examining SRAM in two commercially available embedded micro-controllers (STM32F3 & STM32F4).

Both studies show an increase in the average fHD of intra-die responses as the voltage ramp-up time increases, and there is also an increase in the die-to-die spread, indicating that sensitivity to supply voltage variations is die-dependent. Another study on voltage ramp-up effects by Carnegie Mellon University [17] shows that when compared to a reference bitmap, the noise levels increase with the distance to the ramp-up time in which the reference map was generated, whether higher or lower. This highlights the importance of ensuring a reproducible voltage ramp-up for the SRAM when used as PUF.

Selimis et al. [4] also study the effects of temperature on initialization values, demonstrating that it has a large impact on reliability. Changing the operating temperature raises the average fHD between initializations from around 3% at room temperature (20°C) to around 7% at 60°C or around 14% at -40°C. Again, the spread between dies increases markedly when changing temperatures, demonstrating that response to temperature changes is also strongly die dependent.

3.1.3 Age

The phenomenon of Negative Bias Temperature Instability (NBTI) [18] in MOS-devices cause the SRAM-cells to change their characteristics when held in any state for an extended period of time, and this aging is accelerated when subjected to high voltages and/or temperatures. This poses a problem for SRAM as PUFs since they could change their properties over time, and PUFs must be time-invariant. For this reason, SRAM, which is used as PUF, should ideally not be used to store data during normal operation, as it could affect the PUF response.

This effect has also been used by researchers at Carnegie Mellon University to improve the reliability of SRAM [17] by intentionally reinforcing a particular state during enrollment in a procedure referred to as Directed Accelerated Ageing (DAA). The researchers report achieving a 40% reduction in bit error rate when applying the equivalent of 0.8 years of aging under nominal conditions. They accelerate this aging by operating the circuits at increased voltage and temperature for 120 hours. A drawback of this procedure is that it requires a significant amount of time between manufacture and deployment, as well as specialized equipment for voltage and temperature control, making it a costly procedure when applied at scale.

3.2 PUF Enrollment

Before a PUF is deployed, it goes through an enrollment phase where the challenge/response pairs (CRPs) are recorded into a database [3]. For weak PUFs during this stage, the helper (ECC) data is computed and programmed into the device. If this ECC computation is done off-device, a path must exist to extract the raw PUF data from the device, which could be exploited by an attacker. This path could be destroyed after enrollment by fuses or laser cuts. However, methods have been demonstrated to restore such paths [17].

Using ECC directly to correct errors in PUF potentially introduces leakage into the system, as the ECC helper data could reveal information about the key. To address this issue, a security primitive called a fuzzy extractor [19] is typically used to reconstruct keys from PUFs or other noisy sources such as biometric data. This mechanism allows for the reconstruction of keys from noisy data in such a way that the helper data does not reveal information about the key itself.

While PUFs traditionally rely on error correction to account for variations between initializations, some methods have been proposed to authenticate dies without requiring ECC. Researchers at the University of Massachusetts Amherst [9] have proposed using the expected mismatch in Hamming Distance (HD) between inter- and intra-PUF responses. In the proposed method, the response from the entire population of dies is recorded (referred to as the known fingerprint) in the enrollment phase. During operation, the fingerprint (referred to as latent fingerprint) is compared to each known fingerprint and is assumed to belong to the die which it has the smallest HD to. The authors report achieving an accuracy of 100% when employing this method using a 64-bit fingerprint and 96% when using a 32-bit fingerprint. How many bits are required to identify a population reliably depends on the size of the population and the number of reliable bits in the response.

A similar method has been proposed by researchers at Arizona State University [14]. In this proposed method, the response is accepted if it has a HD less than some threshold. While these methods enable die identification without error correction, they are unsuitable for cryptographic applications as they require a completely stable key to function.

3.3 Fast Erase / Fast Initialization

A limiting factor of SRAM start-up values as identifiers is that the PUF data is only available immediately after powering it on. It is also subject to changes in voltage ramp-up time during startup, which as mentioned in 3.1 can have a significant impact on startup values. Re-initializing the array by powering down the SRAM also takes a prohibitively long time due to data remanence [20], where data remains in volatile memory for some time, even after powering down. To overcome this limitation, a modified SRAM circuit has been proposed by researchers at Arizona State University [21] for use in PUFs, where the cell state {P,Q} can be destabilized to a metastable state {0,0} or {1,1}. This process immediately erases the data in the cell, eliminating any remanence issue, and mimics the state at power-on, meaning the cells can be re-initialized simply by releasing them from this metastable state. The necessary circuit changes for this Fast Erase / Fast Initialization (FEFI) method are area-free and do not affect the normal operation of the memory.

In their original paper proposing this solution, the ASU researchers [21] show that when using this method, the cells have a strong preference for initializing to '1'. However, in subsequent papers by ASU [14], and Intel [22] using this technique, this issue does not seem to appear.

A version of this technology has been implemented at CEA Grenoble [23] as a countermeasure to tamper events by quickly erasing sensitive data. This technology will be important in developing PUF post-processing strategies later in this thesis.

3.4 Existing Commercial Solutions

SRAM PUF IP blocks for ASICs or FPGAs are currently available from Synopsys [24] and have been deployed in commercial products such as Intel's Stratix FPGAs or microcontrollers from NXP. This product is the most similar in operating principle to the implementation discussed in this thesis and will be used as a reference point of comparison.

Other types of PUFs are also available, for example, in the Zynq UltraScale FPGA from AMD [25], which features a ring oscillator PUF design licensed from Verayo (defunct). Maxim Integrated markets their ChipDNA PUF [26], while EMemory offers a "Quantum Tunneling PUF" IP called NeoPUF [27], which is based on comparing transistor pairs to see which has the higher quantum tunneling current. All of these products rely on error correction to account for variations between PUF responses, requiring an enrollment phase before deployment to calculate and store helper data.

Chapter 4

Methodology

This chapter presents the methodology and tools for developing and evaluating PUF post-processing methods. To develop algorithms for reliable PUF key generation, it is necessary to evaluate them using a large set of data to assess their properties under various conditions. A data set of SRAM initializations from a test chip is available and will be used to assess the methods in a known realistic scenario. The proprietary SRAM PUF CyberPUF simulation tool will be used to further evaluate the methods under different conditions, and its general working principle will be explained in this chapter. The combination of measured and simulated data will be used to assess the quality of the key generation methods, however to assess performance (speed) an RTL model will also be implemented.

4.1 Flow

To extract keys from the SRAM array, we will be examining two forms of algorithms using Multiple Evaluation (ME), namely *Temporal Majority Voting* (TMV) and *Memory Cell Classification* (MCC). These key generation methods will first be tested in Matlab/CyberPUF to determine their qualities before being implemented in hardware to determine their clock cycle performance. For functional verification of the PUF module, a pipeline has been established to automatically compare the resulting key in RTL simulation using QuestaSim [28] and the key generated by the corresponding Matlab implementation of the algorithm. Figure 4.1 shows the intended workflow.

The algorithms are tested on a dataset of measurements from a custom ASIC featuring a reference SRAM designed by Synopsys and a modified version of the same memory with the FEFI capabilities described in 3.3. The memories are 128 bits by 128 lines, with sixteen cuts for each type. The available data is taken from a single test chip mounted on a custom board. For each cut, 1000 measurements have been taken at three different voltages at a constant ambient temperature of 25°C. The test chip is manufactured in Global Foundries' 22 nm FDSOI process.

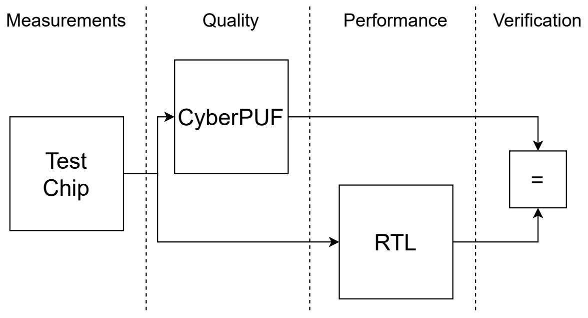


Figure 4.1: Development flow

With the dataset, containing 3000 initializations per cut, we can combine multiple initializations to create a key. Each combination of the same memory section will be referred to as a *trial*. How many trials we can make using the dataset depends on how many initializations are used per trial. For most experiments, each initialization is used only once, without replacement. When this is not the case, it will be made clear. Some experiments are conducted using synthetic data from simulation; again, it will be made clear when this is the case.

4.2 CyberPUF

The CyberPUF tool is a Matlab package developed at CEA Grenoble, which can be used both to characterize measured data from SRAM initializations and to synthesize new data for testing purposes. It also contains functions to asses the quality metrics described in section 2.3.

SRAM initialization data is stored in CyberPUF as three-dimensional matrices with dimensions representing $[initializations, rows, columns]$. From this data, the characteristics of the SRAM array, such as the TMV value of cells and the Hamming Weight, can be calculated. Key data from PUF responses is also stored in three-dimensional matrices with dimensions representing $[trials, keys, bits]$, from which we can use the package functions to asses PUF qualities such as inter- and intra-PUF fHD or auto-correlation.

The package also contains features to synthesize new SRAM data based on given characteristics. This feature is useful for extending the available data and evaluating the tested methods using SRAM which differs from the available data set while having the same statistical properties. To synthesize new initializations of an existing array, the package uses the probability of each cell initializing to '1', combined with noise from a uniform random number generator. The package starts from a distribution of cell probabilities to synthesize new SRAM arrays. Both of these concepts are explained further in the following sections.

Prior to this work, no documentation was produced for the CyberPUF package. Therefore, a description of the package functions used in this work is included in appendix A.

4.2.1 Probability Matrix

The probability of a cell at index l to initialize to '1' can be calculated from equation 2.5, over S initializations and where $b_{s,l,d}$ is the bit value at initialization s of the cell l in device d. By finding this probability for all cells in an array, we can create a probability matrix (P-matrix) that characterizes the array and can be used to simulate the spontaneous initialization of the cells. Figure 4.2 shows an example P-matrix of an array. The matrix is visualized in gray-scale where white cells always initialize to '1' and black cells always initialize to '0'. Grey cells are unstable, and their intensity represents the probability of initializing to '1'. To simulate the spontaneous initialization of the array, we generate a series of uniformly random numbers between zero and one and compare them against the cell probability to determine whether to initialize the cell to '1' or '0'.

$$b_{l,d} = \begin{cases} 1, & RNG(0,1) \le \beta_{l,d} \\ 0, & else \end{cases} \tag{4.1}$$

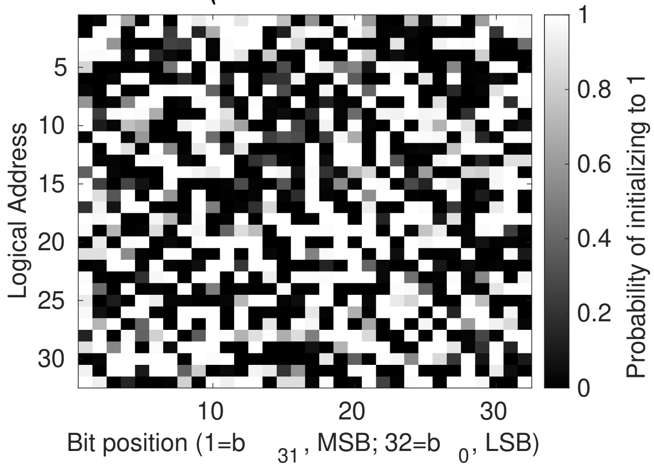


Figure 4.2: Example probability-matrix of a 32x32-bit SRAM array

4.2.2 Probability Distribution

From the probability matrix, we can calculate a distribution of probabilities that characterizes the array, containing a discrete part with cells always initializing to '0' or '1' and a continuous part consisting of cells with a probability $0 < \beta < 1$ of initializing to '1'. Figure 4.3 shows the probability distribution of an SRAM array with discrete components represented as points at the edges of the distribution and a histogram of continuous components aggregated into 50 bins. From this distribution, we can synthesize probability matrices with the same characteristics as measured arrays or arbitrarily designed characteristics that are spatially different.

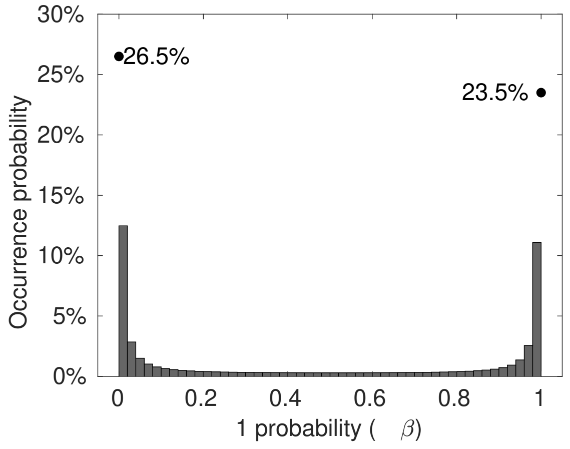


Figure 4.3: Example probability distribution

Synthetic probability-distributions in CyberPUF are characterized by three parameters $[\alpha, \Delta, \delta_p]$:

- α: The proportion of completely stable bits ($\beta = 0$ or $\beta = 1$). Shown as points in figure 4.3.
- Δ: Asymmetry between completely stable bits. The proportion of bits stable at '0' is $(\alpha + \Delta)/2$, and the proportion of bits stable at '1' is $(\alpha - \Delta)/2$.
- δ_p: The proportion of bits that are part of the decaying "bumps" emanating from the edges of the distribution. The integral of the bump emanating from '0' will be $(\alpha + \Delta) \times \delta_p/(2\alpha)$, while the integral of the bump emanating from '1' will be $(\alpha - \Delta) \times \delta_p/(2\alpha)$

The remaining portion of bits $(1 - \alpha - \delta_p)$ is uniformly distributed in the range $0 < \beta < 1$. The example distribution shown in figure 4.3 is synthesized with parameters [0.5,0.03,0.4].

4.2.3 Workflow

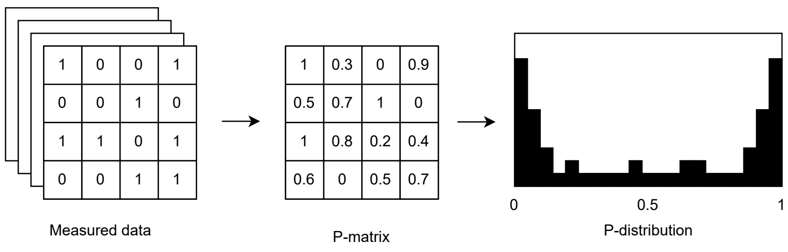


Figure 4.4: Process for characterizing data using CyberPUF

To extract information about the characteristics of a measured data set, it must be imported into Matlab in the CyberPUF format ([initializations, rows, columns]). Depending on how the data is stored, this could be achieved by the CyberPUF function *hexFile2mem* or the standard Matlab *readtable* function. If the data is not compatible with either of these functions, a parser must be written. Once the data is loaded into Matlab, the *calculatepMatrix* CyberPUF function can be used to find the spontaneous initialization probabilities, and *calcPDistribution* can be used to find the characteristics of the data. This process is depicted in figure 4.4.

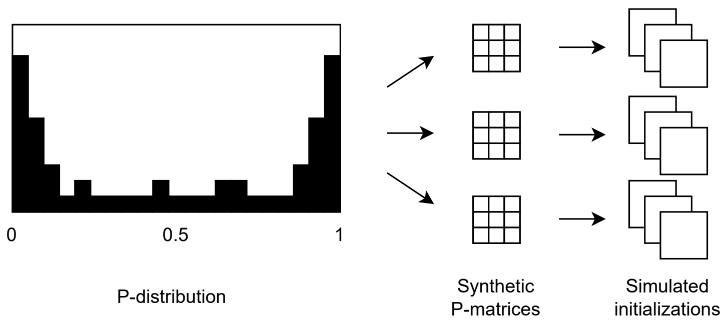


Figure 4.5: Process for synthesizing data using CyberPUF

To create synthetic data, one must obtain a probability distribution using either *calcPDistribution* or *createMixedPDistribution*. From this distribution, one or more probability matrices can be obtained using *getPmatrixFromMixedPDistr*, where each p-matrix represents a unique SRAM array. To simulate spontaneous initializations we use the *simulateMatrices* function. Figure 4.5 shows this process. The resulting data can be used directly within Matlab or exported as hex files using *writeHex*. The resulting hex files can then be used in RTL simulation using QuestaSim.

4.3 Hardware Implementation

While reading SRAM to generate a key could be implemented in software, this would present security issues as an attacker with physical access to the device could execute code to read out the SRAM content and extract the key. Therefore, the section of memory that will be used for key generation is protected from read or write access except when the access is generated by a dedicated hardware key generation module. This module will be implemented as an instruction generator for near-memory computing (NMC) vector co-processor developed at CEA Grenoble [29] referred to as Computational-SRAM (C-SRAM).

Using the C-SRAM co-processor, the PUF data does not have to be transferred over the system bus where potentially malicious devices could be listening. Using a vector processor also allows for the evaluation of many cells in one instruction. Further security measures of the implementation, such as tamper detection and deployment of the key in cryptographic applications, will not be discussed in this work as they are beyond the scope of this thesis. The PUF module is integrated into the C-SRAM as a privileged instruction generator that can generate read operations to sections of the memory not accessible by externally generated operations, visualized in figure 4.6. To initiate the key generation procedure, the CPU must send a PUF key generation instruction to the C-SRAM.

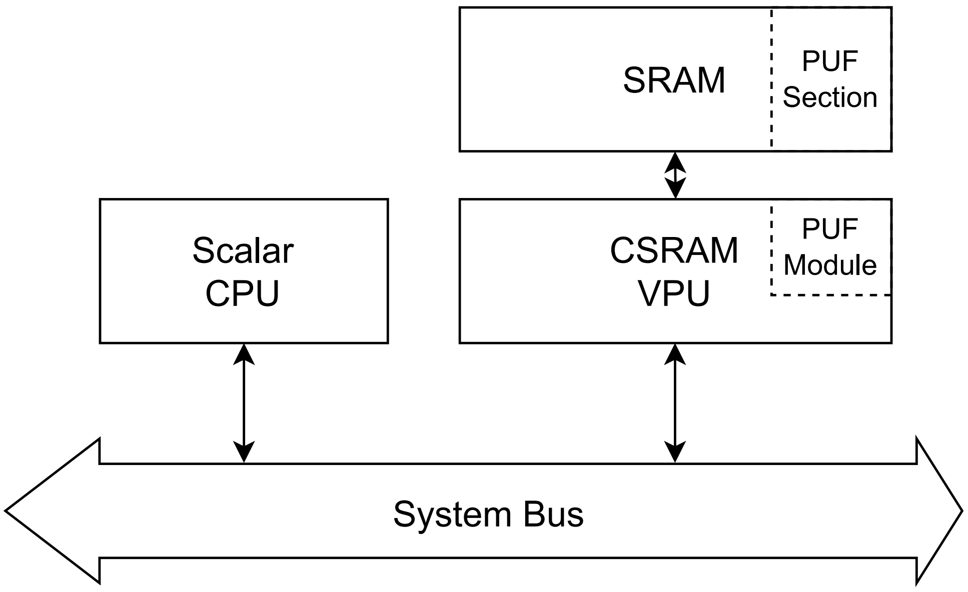


Figure 4.6: System architecture

The C-SRAM is a SIMD-like vector processor that features a six-stage pipeline and a 128-bit architecture that can process 32, 16, or 8-bit words. It uses a custom ISA with instructions for accelerating security primitives/algorithms such as AES. The key benefit of using the NMC architecture is reduced traffic on the system bus, which saves both clock cycles and energy. The C-SRAM also has a wide 128-bit connection to its memory, meaning it needs to make fewer requests to the memory to access the same data compared to using the much narrower system bus.

4.3.1 RTL Simulation

The C-SRAM project already has an established RTL simulation environment using QuestaSim, which is also used in this work. To simulate the spontaneous initialization of the SRAM array, the environment has been extended to detect re-initialization events and load a hex file into the memory. This is accomplished using the commands *when* and *mem load* [30].

When is used to detect events in simulation, in our case, a signal from the PUF FSM which tells the SRAM to re-initialize. Once such an event is detected, *mem load* is used to fill the memory with data dumped from CyberPUF using *writeHex*. Another flag in the FSM signals that it has finished, at which point the key is extracted using *examine*. The key is then automatically compared to the key produced in Matlab using the same input to verify that the RTL is correct.

Chapter 5

Experimental results

As previously discussed, there is significant noise present between different SRAM initializations, requiring helper functions in order to stabilize the PUF response. From this stabilized PUF response, which consists of a large array of data, a key/identifier of some fixed length must be extracted.

5.1 Quality

5.1.1 Nominal

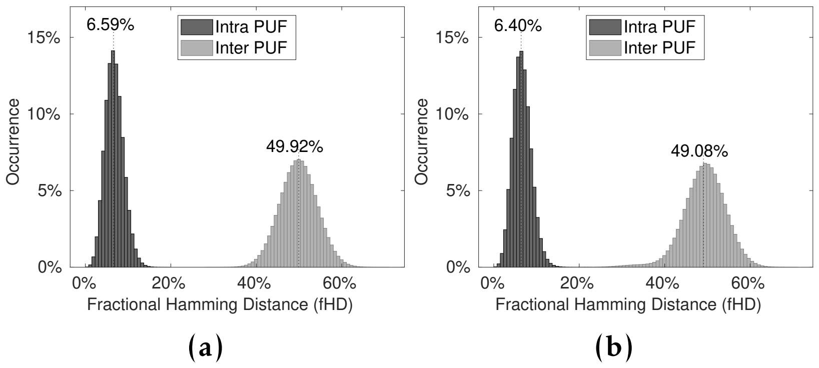


Figure 5.1: Nominal fHD of reference memory (a) and custom memory (b)

As a baseline to compare the quality of these helper functions against, we examine the inter- and intra-PUF fHD of the test memories without any helper functions when considering each 128-bit line of memory a unique PUF, shown in figure 5.1. The golden bitmaps are created from the TMV over all initializations. As can be seen from this figure, the custom memory has a slightly lower average intra- and inter-PUF fHD, which is likely related to the average Hamming Weight, which is 0.48 for the reference and 0.43 for the custom memory.

The probability distributions show that the custom memory has more bits that are stable at '0' and slightly fewer bits that are stable at '1' compared to the reference. Both memories consistently show more stable bits at '0' than '1'. Figure 5.2 shows the probability distribution of the first cuts of both memories, which are representative of the types. The average cell probability for the reference memory is 48.2% and 45.6% for the custom memory.

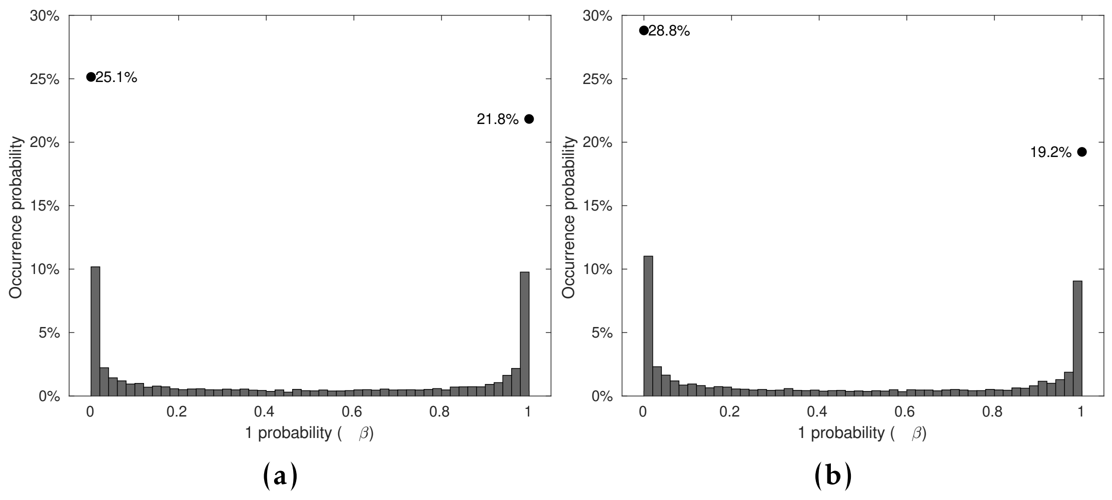


Figure 5.2: P-distribution of reference memory (a) and custom memory (b)

	Reference memory				Custom memory			
	0.7V	0.8V	0.9V	Mixed	0.7V	0.8V	0.9V	Mixed
Intra-PUF	6.17%	6.10%	6.01%	6.59%	5.58%	5.81%	5.57%	6.4%
Inter-PUF	49.94%	49.90%	49.99%	49.91%	47.07%	48.29%	47.47%	47.57%

Table 5.1: Nominal average fHD

When comparing measurements taken at different voltages (shown in tables 5.1, 5.2 & 5.3), we observe that there is little difference between PUF quality metrics at the different voltages. The reference memory displays better qualities across all metrics, deviating the least from the ideal $\frac{1}{2}$, likely due to the better average Hamming Weight.

	Uniformity			Aliasing			Auto-correlation		
	Min.	Avg.	Max.	Min.	Avg.	Max.	Min.	Avg.	Max.
0.7V	0.34	0.48	0.65	0.46	0.48	0.52	0.49	0.50	0.51
0.8V	0.33	0.48	0.63	0.45	0.48	0.51	0.49	0.50	0.51
0.9V	0.34	0.48	0.66	0.45	0.48	0.51	0.49	0.50	0.51
Mixed	0.33	0.48	0.65	0.45	0.48	0.51	0.49	0.50	0.51

Table 5.2: Nominal quality metrics for reference memory

	Uniformity			Aliasing			Auto-correlation		
	Min.	Avg.	Max.	Min.	Avg.	Max.	Min.	Avg.	Max.
0.7V	0.12	0.41	0.61	0.39	0.41	0.44	0.46	0.47	0.49
0.8V	0.16	0.46	0.65	0.42	0.46	0.48	0.47	0.48	0.51
0.9V	0.13	0.43	0.62	0.40	0.43	0.46	0.46	0.47	0.49
Mixed	0.13	0.43	0.61	0.41	0.43	0.46	0.46	0.47	0.49

Table 5.3: Nominal quality metrics for custom memory

5.1.2 Temporal Majority Voting

A conceptually trivial method of reducing noise is simply finding the temporal majority value for each cell. This requires re-initializing the array an odd number of times and keeping the value that appears most frequently. It is also possible to impose minimum requirements for cell stability in order to qualify for use as key material. In [15], the authors demonstrate how increasing TMV voting windows reduce intra-PUF Hamming Distance. This form of voting does, however, require counters for each evaluated cell to keep track of initialization values, which can be implemented in software as in [15] or in hardware as in [22]. A hardware implementation is preferable in terms of speed but is also costly as it requires additional circuitry. A software implementation is more cost-effective but has significant performance overhead as the evaluation of each cell requires operations beyond simple bitwise operations.

To reduce the complexity of calculating which value appeared most frequently in a given number of initializations, we choose a voting window equal to $2n-1$, where n is some positive integer. This way, when we use an n-bit counter, the TMV value of the cell will be equal to the value of the MSB in the counter.

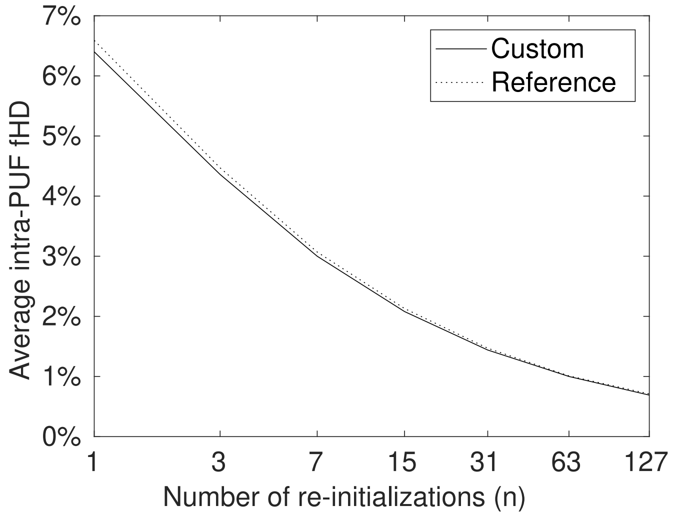


Figure 5.3: Average intra-PUF fHD with TMV (mixed voltage)

Figure 5.3 shows how increasing the TMV voting window reduces the average intra-PUF fHD from around 6.5% without TMV down to around 0.7% using 127 re-initializations. The results show diminishing returns as the voting window increases, from an improvement of more than one percentage point when going from TMV3 to TMV7 to an improvement of less than half a percentage point when going from TMV63 to TMV127. It is, however, noteworthy that at TMV127, the average fHD is less than 1/128, meaning in a 128-bit identifier, one can expect an average Hamming Distance of less than one.

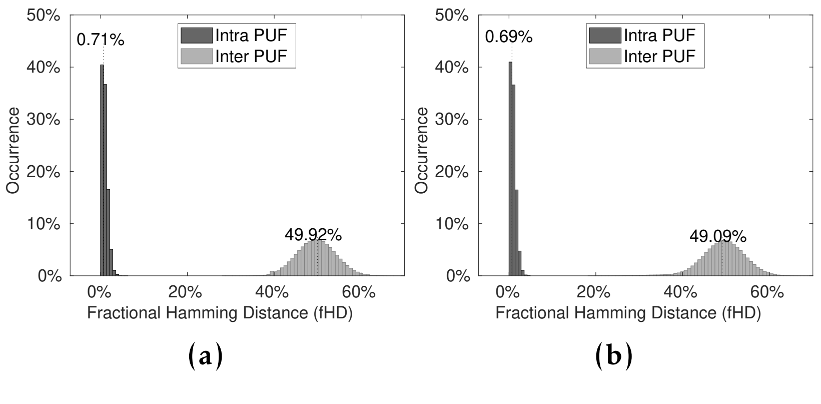


Figure 5.4: fHD of reference memory (a) and custom memory (b) with TMV127

By using each 128-bit line of memory as a unique PUF, we can evaluate this theory. Figure 5.4 shows that the most commonly occurring intra-PUF fHD when using each 128-bit line as an identifier is 0, which occurs around 40% of the time for both memory types.

Uniformity			Aliasing			Auto-correlation		
Min.	Avg.	Max.	Min.	Avg.	Max.	Min.	Avg.	Max.
0.13	0.43	0.60	0.40	0.43	0.46	0.46	0.47	0.49

Table 5.4: Quality metrics for custom memory using TMV127

Table 5.4 shows that the quality metrics of the custom memory using TMV127 is nearly identical to the nominal metrics (table 5.3) when using mixed voltage. The averages are across the board lower than ideal, and in the case of uniformity, the minimum is concerningly low. With these metrics, it is likely necessary to implement some kind of privacy amplification to produce stronger keys.

5.1.3 TMV With Stability Requirements

As previously mentioned, TMV also allows imposing minimum stability requirements on cells, which excludes cells that are too unstable from being used as key material. When using $2^n - 1$ votings, we can simplify the comparison into a bitwise operation by examining if the first m MSBs are equal. For example, if four bits are used to count (TMV15) and the first two MSBs are '1', then the cell initialized to '1' at least 12/15 times, and if both are '0', then the cell initialized to '1' at most 3/15 times. Expanding the number of bits to compare (m) will increase the threshold for stability required to be used as key material. When $m = n$ the cell would have to always initialize to the same value in order to be considered as key material, this case will be considered separately in section 5.1.4.

Using an n bit counter ($2^n - 1$ votings) and comparing the first m bits for equivalence, we can get the following threshold p_{Th} to accept a cell as key material where cells in the range $1 - p_{Th} < \beta < p_{Th}$ are rejected:

$$p_{Th} = \frac{2^n(1 - 2^{-m})}{2^n - 1} \qquad \{n, m\} \in \mathbb{N} > 1; m < n \tag{5.1}$$

Table 5.5 shows the tested cases and their stability requirements.

n\m	2	3	4	5	6
4 (TMV15)	80%	93.3%	-	-	-
5 (TMV31)	77.4%	90.3%	96.8%	-	-
6 (TMV63)	76.2%	88.9%	95.2%	98.4%	-
7 (TMV127)	75.6%	88.2%	94.5%	97.6%	99.2%

Table 5.5: Stability requirement p_{Th} for n bit counter and m bit equivalence

Since this process excludes some bits from being used as key material, it is also necessary to define some method of assembling the remaining bits into an identifier. A simple method of creating an identifier of some fixed length L is to just take the first L stable bits found. A drawback of this method is that misidentifying a bit as stable or unstable will shift all subsequent bits left or right by one position, meaning we can expect a larger spread in intra-PUF fHD compared to methods where a miss identification only affects one bit. Testing shows that this method creates such a large spread in the intra-PUF fHD that the average is not comparable to the other discussed methods.

Instead, we can define a method of column-wise selection, where we keep the first bit in each column to meet our stability criteria. This way, a miss classification of a cell only affects one column. Table 5.6 shows the average inter-PUF fHD when applying this method. Results show that increasing the stability threshold decreases the reliability, and this method performs worse than TMV alone, regardless of the threshold.

n\m	2	3	4	5	6
4 (TMV15)	2.70%	3.76%	-	-	-
5 (TMV31)	1.85%	2.25%	3.49%	-	-
6 (TMV63)	1.30%	1.56%	2.07%	2.68%	-
7 (TMV127)	0.90%	1.10%	1.40%	1.92%	2.95%

Table 5.6: Average intra-PUF fHD using TMV + p_{Th}

5.1.4 Memory Cell Classification

As mentioned in section 5.1.2, there is a special case where only cells that are completely stable are accepted as key material. While this can be achieved by counters like in regular TMV, it turns out that it is not necessary to keep track of how many times each cell initialized to ’1’. Researchers from the University of Sevilla [31] have presented a method by which cells can be labeled as stable or unstable without counting, referred to as memory cell classification.

In this method, visualized in figure 5.5, the first initialization of the cells is recorded, and all subsequent initializations are compared to the first. All cells have a label that is initially "stable" (’S’), and if the comparison with the first initialization is ever a mismatch, the label is updated to "unstable" (’U’). During key generation, we only consider cells that are labeled ’S’. This method requires only bitwise operations, meaning a large set of cells can be evaluated at once and requires memory, which is equal to twice the number of cells that are being evaluated. A drawback of this method is that a large number of cells is likely to be excluded from use as key material, meaning more memory is needed to find enough key material.

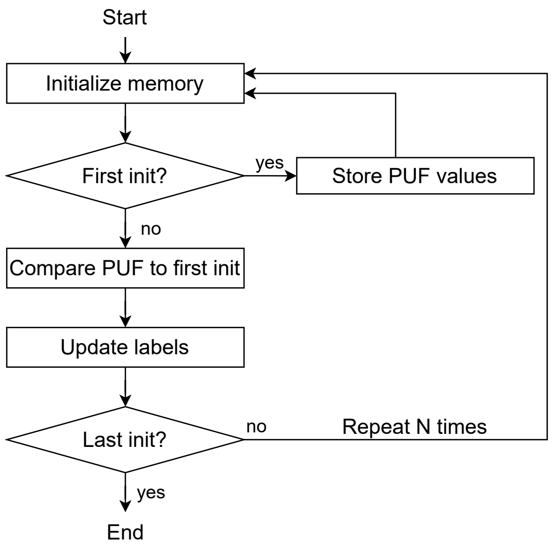


Figure 5.5: Process for memory cell classification as described in [31]

Again, we require methods of assembling a key from the cells labeled 'S'. Like in TMV, simply using the first L stable bits as the key would result in a huge spread in the intra-PUF fHD, as each misclassification would affect all subsequent bits. We, therefore, again turn to column-wise bit selection, by which we achieve the results shown in figure 5.6. These results continue the trend shown in TMV with stability thresholds, where higher thresholds are less reliable.

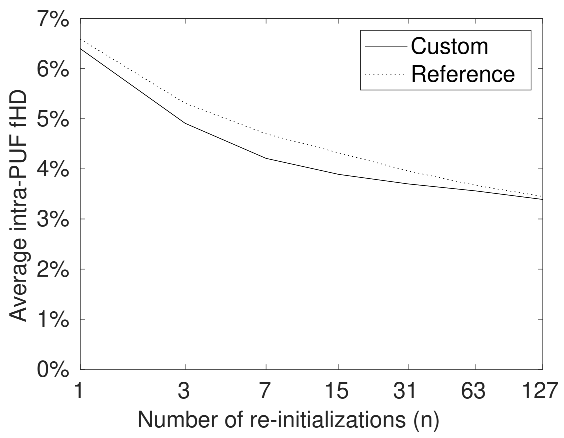
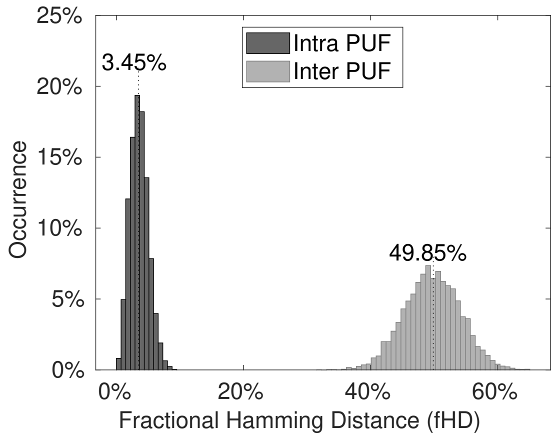
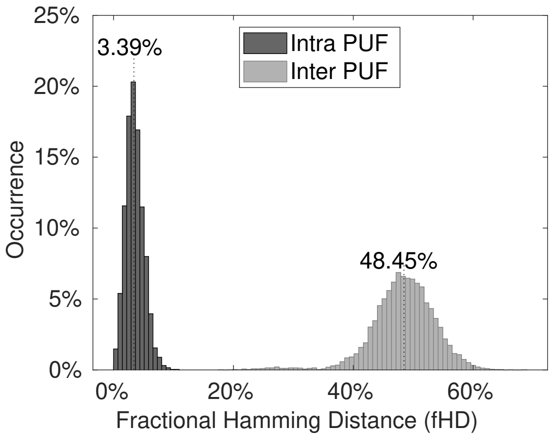


Figure 5.6: Average intra-PUF fHD when using memory cell classification & column-wise bit selection (without replacement, mixed voltage)

Interestingly the Intra-PUF fHD (figure 5.7) appears less Gaussian than nominal and TMV results, though it is unclear what the reason for this is.



(a)



(b)

Figure 5.7: fHD of reference memory (a) and custom memory (b) with MCC127 & column-wise bit selection

The quality metrics for MCC, shown in table 5.7, are worse than the nominal quality metrics and of TMV, with lower averages and minimums. Using this method, some form of privacy amplification to ensure strong keys is even more necessary.

Uniformity			Aliasing			Auto-correlation		
Min.	Avg.	Max.	Min.	Avg.	Max.	Min.	Avg.	Max.
0.11	0.41	0.57	0.30	0.41	0.54	0.44	0.47	0.59

Table 5.7: Quality metrics for custom memory using MCC127 & column-wise bit selection

While these results may be worse than TMV at an equivalent number of initializations, they also show that it is significantly less computationally complex and allows the evaluation of more cells per instruction. The exact difference in terms of time to generate the key will be discussed in section 5.2. Because the time to generate the key is likely lower at an equivalent number of initializations we can use a larger number of initializations compared to TMV without increasing the time necessary to create the key. So far, all results have been obtained by using each measured initialization from the dataset only once. However, when using large windows, it becomes necessary to use each one multiple times. In the following results, we therefore choose at random n initializations taken from the 3000 available, with replacement.

Results from this experiment, shown in figure 5.8, demonstrate that even at 2047 initializations, this method is still less reliable than TMV at 127 initializations. Unlike in TMV the results improve at an accelerating rate when the number of initializations increase by an order of magnitude, however since this is not the case when using small windows it seems likely that this is just an artifact of using the dataset with replacement.

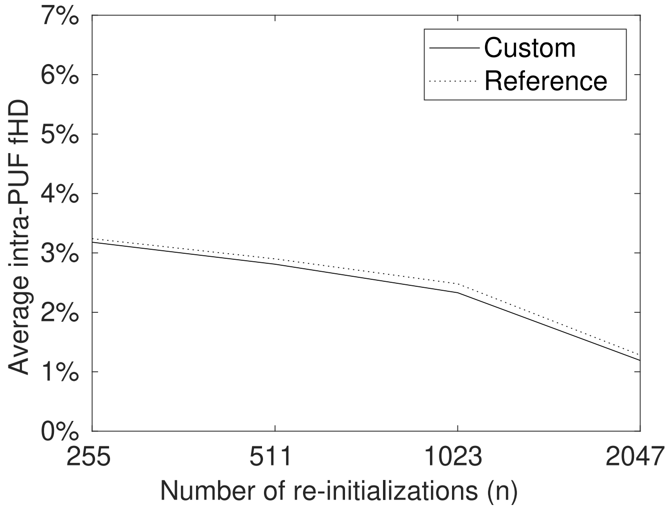


Figure 5.8: Average intra-PUF fHD when using memory cell classification & column-wise bit selection (with replacement, mixed voltage)

Again the intra-PUF fHD (figure 5.9) is less Gaussian than in MCC and nominal results. The method produces the expected result with no error around 20% of the time for both memories.

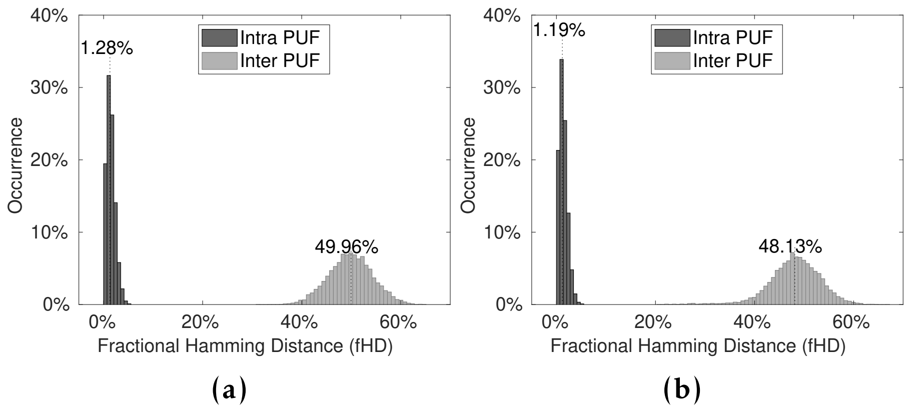


Figure 5.9: fHD of reference memory (a) and custom memory (b) with MCC2047 & column-wise bit selection

The quality metrics for MCC, shown in table 5.8, seem to improve when using a larger voting window, both in terms of averages and minimums.

Uniformity			Aliasing			Auto-correlation		
Min.	Avg.	Max.	Min.	Avg.	Max.	Min.	Avg.	Max.
0.34	0.44	0.55	0.39	0.47	0.57	0.45	0.50	0.54

Table 5.8: Quality metrics for custom memory using MCC2047 & column-wise bit selection

An interesting question regarding this method is how many lines of memory are required to find enough key material. By selecting bits column-wise, we are relying on finding at least one stable bit in each column and are likely discarding a large number of stable bits as we already selected a bit in that column. In a memory cell array where a proportion α of cells are stable, we can assume that the probability of any randomly sampled cell being stable is also α. When looking at L columns, we could estimate to find $L \cdot \alpha$ new bits for each row and requiring $1/\alpha$ lines to fill all columns. However, this does not account for the chance of new bits being found in columns that are already filled. Instead, we can estimate the probability $P_\emptyset$ of having found no stable bits in a column after examining n rows to be:

$$P_\emptyset = (1-\alpha)^n \tag{5.2}$$

And when considering L number of columns, we can estimate the number of columns $C_\emptyset$ where no stable bits have been found after n rows to be:

$$C_\emptyset = L \cdot (1-\alpha)^n \tag{5.3}$$

As n increases, $C_\emptyset$ will tend towards but never reach zero. Since columns being filled or not is a discrete event, we can round down to zero once we go below one in order to find the number of lines where we expect to have filled all columns, or Lines To Key (LTK):

$$LTK_{expected} = \frac{-log(L)}{log(1-\alpha)} \tag{5.4}$$

Since all the available data sits somewhere around 55% stable bit probability (α), we use CyberPUF to simulate SRAM arrays with lower stable bit probability. Figure 5.10 shows a scatter of simulated 256×128-bit arrays with parameters $[\alpha, 0, \alpha/2]$ and the model as a dashed line. In these simulations, the model seems to give a good picture of how many lines of memory will be required on average to produce a key.

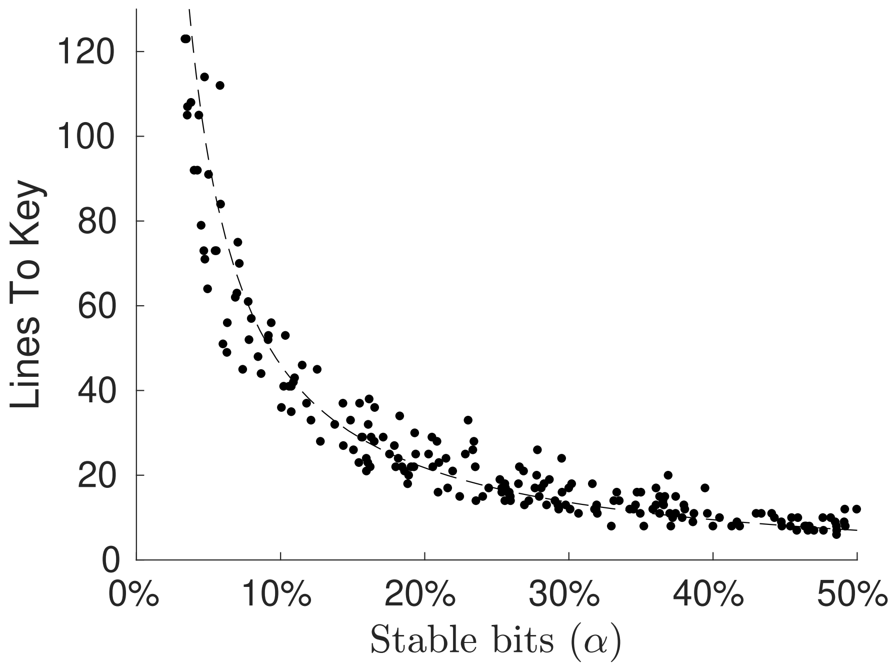


Figure 5.10: Expected lines to key vs stable cell probability

The expected Hamming Weight of this method is the number of bits stable at '1' divided by all stable bits. As shown in figure 5.2 both memories have more bits stable at '0' than '1'. For the reference memory, this proportion comes out to 0.47, and for the custom memory, it is 0.43. The actual average HW produced by the algorithm is 0.47 and 0.41 for the reference and custom memory, respectively.

5.1.5 Column-wise Parity

The fact that Memory Cell Classification using column-wise bit selection requires an unknown amount of memory to produce a key can pose a challenge, as it is hard to predict how long it will take to produce a key or even if an array contains enough stable bits to produce a key or not. Therefore, we propose another method of assembling our labeled cells into a key, namely column-wise parity. In this method, we utilize the labels themselves instead of the underlying cell value by calculating the column-wise parity of the labels in some fixed number of rows. We refer to the memory section that is considered as a page, with a page size of n rows. Again, the probability of any cell being stable is α, and stable cells are labeled with a '1' while unstable cells are labeled with '0'. The probability of a column parity being '1' is the cumulative probability of finding an odd number of stable cells in n rows or the complement of the cumulative probability of finding an even number of stable cells:

$$HW_{expected} = 1 - \sum_{k=0}^{\lfloor n/2 \rfloor} \binom{n}{2k} \alpha^{2k} (1-\alpha)^{n-2k} \qquad n \in \mathbb{N} \tag{5.5}$$

Using this method, we require larger pages the further α is from 50% in order to have an expected Hamming Weight of 0.5. Figure 5.11 shows the expected HW vs page size for five different α. The results are symmetric around 0.5 with respect to the difference $0.5 - \alpha$, so, for example, $HW_{expected}(\alpha = 0.6) = 1 - HW_{expected}(\alpha = 0.4)$.

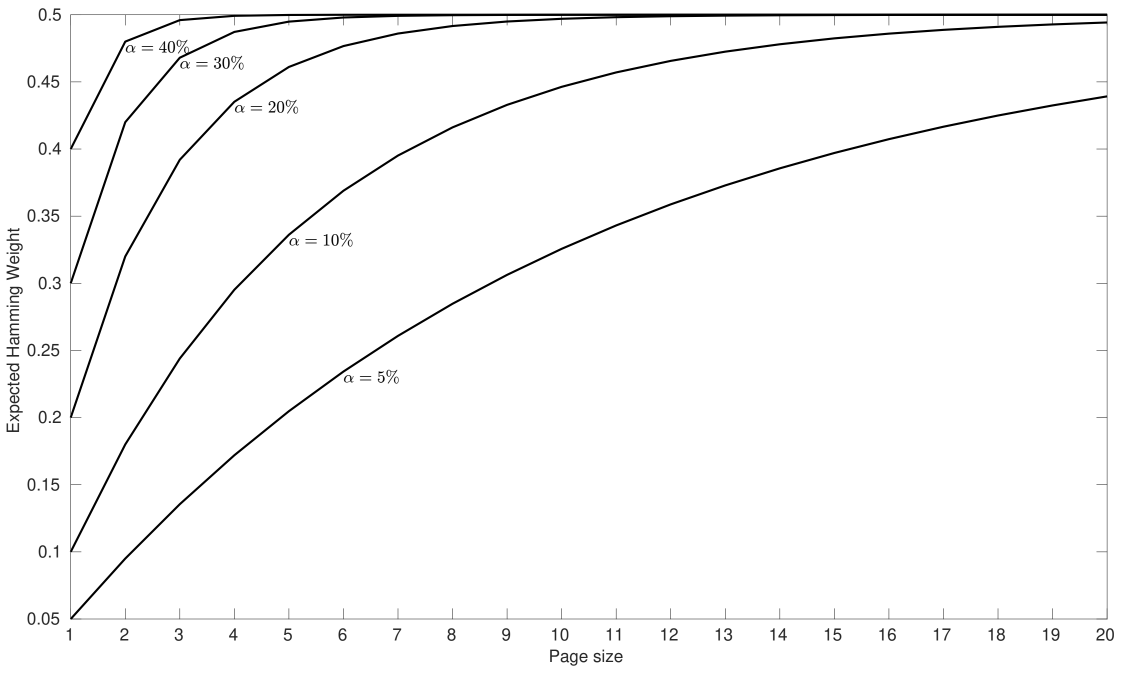


Figure 5.11: Expected Hamming Weight vs page size for different α

Page \n	255	511	1023	2047
2	7.45%	6.50%	5.25%	2.55%
4	13.73%	12.09%	9.91%	4.96%

Table 5.9: Average intra-PUF fHD (reference memory) using MCC and column-wise parity

Results using column-wise parity (table 5.9) shows an increase in inter-PUF fHD compared to column-wise bit selection, but also an improved intra-PUF fHD (figure 5.12). The average Hamming Weight also increases compared to column-wise bit selection, from 0.47 to 0.50 in the reference memory and from 0.41 to 0.495 in the custom memory when using a two-line page. For memories that have a strong skew in stable cells towards ’1’ or ’0’, this method could be effective at increasing the entropy. However, increasing the page size, which is needed to increase HW, also decreases reliability meaning this method is most suited for memories with an α near 50%.

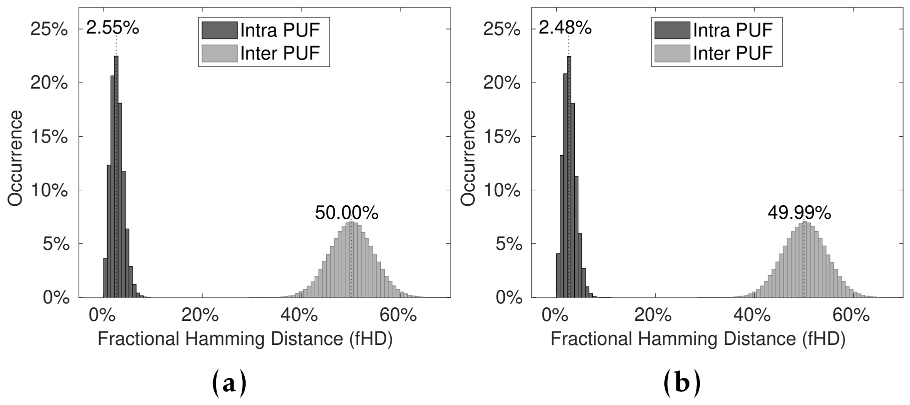


Figure 5.12: fHD of reference memory (a) and custom memory (b) with MCC2047 and column-wise parity (page size 2)

	Uniformity			Aliasing			Auto-correlation		
PageSize	Min.	Avg.	Max.	Min.	Avg.	Max.	Min.	Avg.	Max.
2	0.38	0.50	0.63	0.47	0.50	0.54	0.49	0.50	0.56
3	0.38	0.50	0.61	0.42	0.50	0.55	0.47	0.50	0.52
4	0.38	0.50	0.62	0.44	0.50	0.55	0.49	0.50	0.52

Table 5.10: Quality metrics for custom memory using MCC2047 & column-wise parity

The quality metrics using this method (table 5.10) are significantly improved compared to both of the previous methods. The averages are ideal even using a page size of only two, and minimums & maximums are also generally better.

5.2 Hardware Implementation

5.2.1 Memory Cell Classification

For the hardware implementation, we begin with the Memory Cell Classification method, as it is the simplest to implement. The algorithm (algorithm 1) requires two helper memory sections, *map*, and *copy*, where both are of the same size as the section of memory we want to analyze. *Copy* stores the first initialization of the PUF, to which all subsequent initializations will be compared, while *map* stores the labels for stable cells ('S') or unstable cells ('U') for all cells. For simplicity, we use '0' to denote 'S' and '1' to denote 'U' and initialize *map* to '0'. This makes it simple to update the labels with the result of the comparison without needing to invert.

Algorithm 1 Memory Cell Classification:

```
Initialize PUF
for $i \in [1...size]$ do
    $map_i \leftarrow 0$
    $copy_i \leftarrow PUF_i$
end for
for $r \in [1...n]$ do
    Initialize PUF
    for $i \in [1...size]$ do
        $map_i \leftarrow (PUF_i \oplus copy_i) \vee map_i$
    end for
end for
```

The C-SRAM pipeline (figure 5.13) accesses memory in three of its stages (DEC, RD1 & WB) but has only a single port memory. Therefore, it is not possible to issue new instructions each cycle, and any interleaving must be carefully aligned to avoid attempting two memory accesses in one cycle. This structural hazard is highly limiting in terms of performance as new instructions can, at most, be issued every three cycles. To partially address this issue, the C-SRAM features an internal register of the same width as the memory lines (128-bit), which can be used as a source and/or destination for operations, reducing the number of memory accesses per instruction and allowing more efficient use of the pipeline. Using the internal register for intermediate results reduces the required cycles per PUF line from six to four.

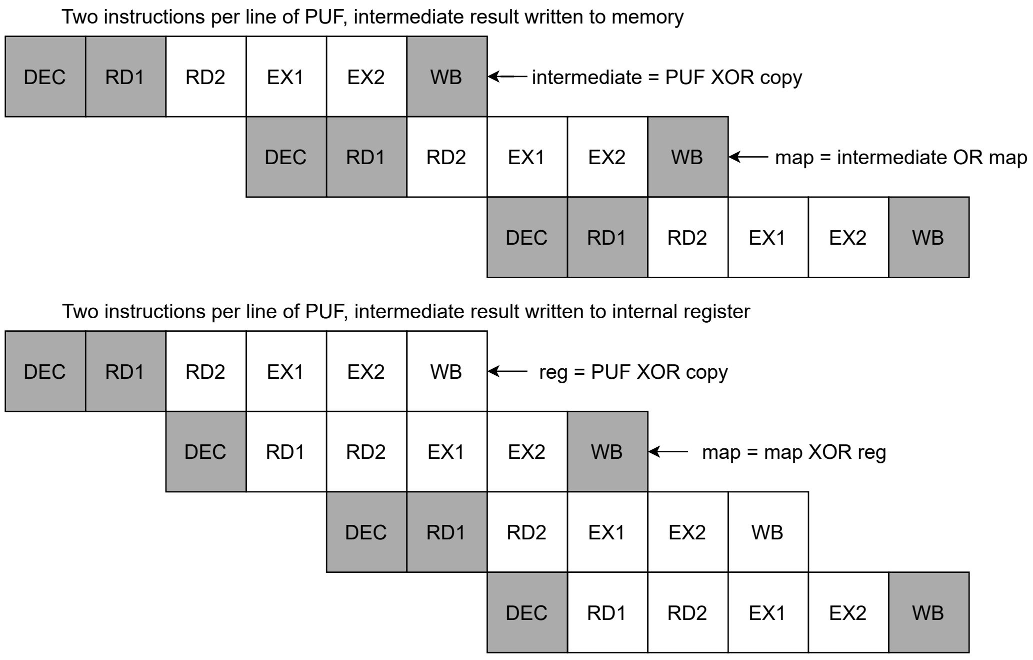

Two instructions per line of PUF, intermediate result written to memory
DEC
RD1
RD2
EX1
EX2
WB
intermediate = PUF XOR copy
map = intermediate OR map
Two instructions per line of PUF, intermediate result written to internal register
reg = PUF XOR copy
map = map XOR reg

Figure 5.13: Instruction alignment in pipeline

It turns out, however, that there is another way of optimizing the number of cycles required per line. The C-SRAM has the ability to perform a "masked" write to the memory, where only the bit positions specified in the mask are affected. This effectively implements the following logical function:

$$dest \leftarrow ((src_1\ OP\ src_2) \wedge mask) \vee (dest \wedge \neg mask)$$

Where OP can be any of the available operations in the C-SRAM (bit-wise or not). By using the result of the operation as the mask and data at the same time, we can use this feature to implement in a single instruction the logical function:

$$dest \leftarrow (src_1\ OP\ src_2) \vee dest$$

This change requires only a small change to the write-back stage of the C-SRAM and brings the number of cycles required per PUF line down to three. Performing Memory Cell Classification on p lines of PUF using n initialization then takes approximately $n(3p + t)$ clock cycles, where t is the number of cycles required to re-initialize the memory.

To assemble the labeled bits into a key through column-wise bit selection, we use a supporting variable $follow$ which keeps track of which columns have been filled. This process is described in algorithm 2.

Algorithm 2 Column-Wise Bit Selection:

```
$follow \leftarrow 0$
$key \leftarrow 0$
$i \leftarrow 1$
while $follow \neq 0xff..ff$ do
    $key \leftarrow (\neg(map_i \vee follow) \wedge copy_i) \vee key$
    $follow \leftarrow map_i \vee follow$
    $i \leftarrow i + 1$
end while
```

The column-wise bit selection process takes $14 \cdot l$ clock cycles, where l is the number of lines required to find a key. The total time needed to find the key is dominated by the Memory Cell Classification for a realistic number of initializations (n), and we, therefore, want to optimize the parameters of this process. The number of cycles this process takes depends strongly on the size of the memory that is being examined.

As was shown in figure 5.10, we can expect that only a small portion of the memory is necessary to find enough stable bits to produce a key. Therefore, we don't have to perform the Memory Cell Classification on the entire 128-line memory. However, evaluating the memory one line at a time will lead to high overheads in re-initializing the memory. We, therefore, divide the memory into pages of p number of lines, where if a key is not found within a page, we move on to the next.

Figure 5.14 shows the number of clock cycles needed to find a key for combinations of lines needed to create the key (l) and various page sizes with and without FEFI (assuming 100 CCs to re-initialize without FEFI). When using FEFI, it is, in most cases, better to use a small page size as the overhead of re-initializing memory is rather negligible. However, without FEFI, it is better to use a medium or large page size as the cost of re-initializing is substantially higher.

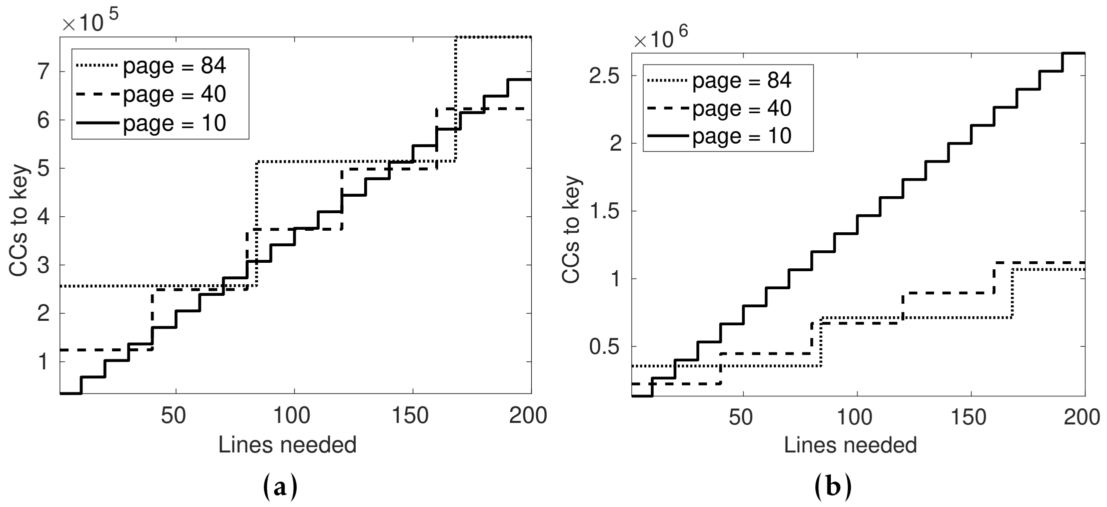


Figure 5.14: Clock cycles to key using MCC with (a) and without (b) FEFI

5.2.2 Temporal Majority Voting

To implement TMV in the C-SRAM, we need to arrange the counters in memory in such a way that we can utilize the vectorialization functionality optimally. The minimum data size of the C-SRAM is 8 bits, so we will be using this size for the counters. However, the method could be extended to larger counters at the cost of performance.

Using 8-bit counters, we require 8 lines of memory for each line of PUF we evaluate, and we can count up to 255 initializations. In the 128-bit C-SRAM, we can evaluate 16 cells at once. The counters will be arranged such that the first line of helper memory stores the counters for each eighth bit starting from one ($b_1, b_9, ...$); the second line stores the counters for each eighth bit starting from two ($b_2, b_{10}, ...$) and so on. This arrangement is shown in figure 5.15 .

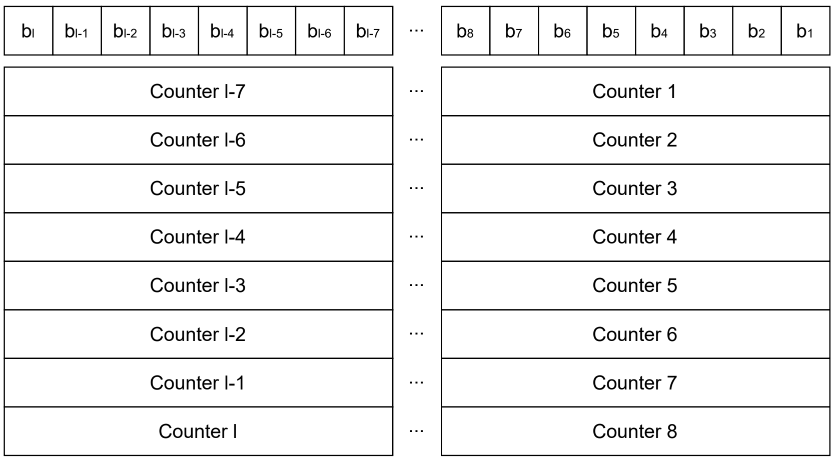


Figure 5.15: TMV counter arrangement

These groupings of cells will be evaluated together. To perform the counting, described in algorithm 3, we shift the PUF line so that the cells we are evaluating are located in the first bit of each byte in the 128-bit word and mask out all other bits. Each byte will now contain a word representing either one or zero and can be added directly to the counters.

Algorithm 3 Temporal Majority Voting:

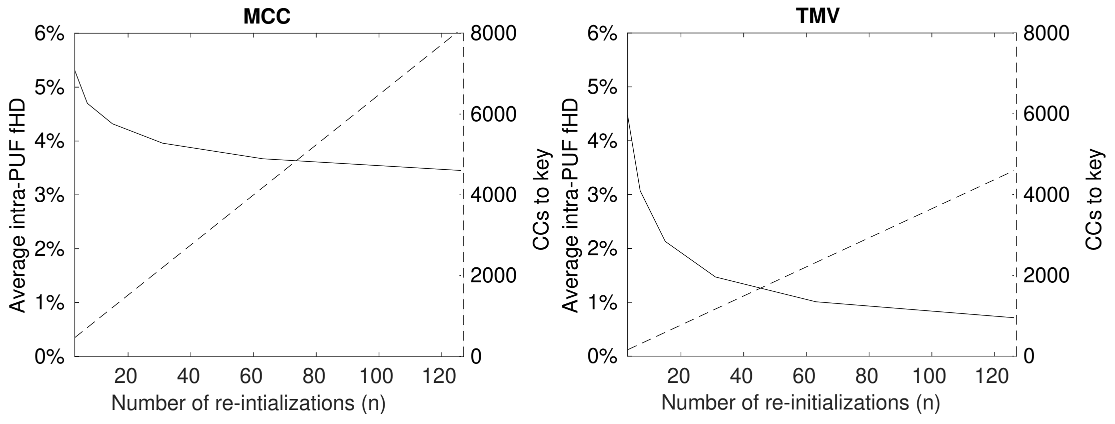
```
for $i \in [0...7]$ do
    $counters_i \leftarrow 0$
end for
for $r \in [1...n]$ do
    Initialize PUF
    for $i \in [0...7]$ do
        $counters_i \leftarrow counters_i + ((PUF \gg i) \wedge 0x01..01)$
    end for
end for
```

When using the C-SRAM, we require three instructions per line of counters, and if using the internal register to store intermediate results, we need four cycles between starting each line. With eight lines of counters, this means we require 32+t clock cycles per initialization, where t is the number of cycles to reinitialize the memory. This holds for up to 255 initializations, which is the maximum we can count using 8-bit data size. To go beyond this would require expanding to 16-bit counters meaning only half as many cells can be evaluated per instruction, which would double the number of cycles required per initialization.

5.3 Quality and Performance Comparison

With the quality metrics extracted using CyberPUF and the clock cycle performance from the hardware implementation, we can compare the proposed methods. The quality metrics for temporal majority voting have been demonstrated to be superior to memory cell classification at an equivalent number of initializations. However, it was previously hypothesized that using memory cell classification would be faster for the same number of initializations since more cells can be evaluated simultaneously. But on the contrary, results from the hardware implementation show a lower number of cycles being required for TMV compared to MCC. The reason for this is that more memory needs to be examined in MCC since we exclude a large number of cells from being used as key material.

Implementation on the C-SRAM is severely hampered by the use of single port memory preventing issuing new instructions at every clock cycle. Since the co-processor accesses memory in three of its six stages, using three-port (2R1W) memory could improve the performance of MCC by nearly a factor of three. TMV would not see the same gain with this bandwidth upgrade since it is more limited by the number of required instructions than by the number of memory accesses. Even with this speedup in MCC, it is not so clear that it would be preferable over TMV since reliability at the same number of initializations is significantly worse. Figure 5.16 shows the quality and clock cycle performance of the current implementations for a range of re-initializations.



(a) (b)

Figure 5.16: Intra-PUF fHD & Clock cycles to key (Reference memory)

There are few available implementations to compare against which list the clock cycle performance of their solutions. However, one point of comparison is the PUFKY ring-oscillator PUF [32]. In their paper, the authors state that their designed PUF takes 4.59ms to produce a 128-bit output (regardless of clock speed) and 55k cycles to perform error correction/fuzzy extraction. To do TMV with 127 initializations, the proposed implementation takes 4623 clock cycles. With the C-SRAM operating at 100 MHz this means it would take 46.23μs to produce a 128-bit key.

But the most relevant point of comparison would be the Synopsys PUF IP, which is SRAM-based like our implementation. Synopsys does not currently publish time to key numbers. However, their PUF IP originates from their acquisition of Intrinsic ID, which previously marketed the product QuiddiKey-100. In a product brief for this product [33], Intrinsic-ID states that the *time to root key* is 49k-68k cycles, which likely includes error correction/fuzzy extraction. In terms of reliability, the proposed solution is not comparable to implementations using error correction, which can likely attain reliability that is better by many orders of magnitude. However, as previously discussed, this requires an enrollment phase, which can be costly and potentially present security issues.

Chapter 6

Conclusion

In this thesis, we have examined the TMV and MCC methods of improving the reliability of SRAM-based Physical Unclonable Functions to investigate the feasibility of eliminating the need for an enrollment phase. Out of the methods for PUF response stabilization examined in this thesis, it is clear that Temporal Majority Voting provides more reliable results and better performance. Both methods tested demonstrate good average quality metrics but can, in some cases, produce unbalanced keys with bad uniformity. To use these methods for security purposes, it is likely necessary to employ some form of privacy amplification to increase the strength of the keys.

Within the voting windows which could be tested with the available dataset, temporal majority voting produced no error around 40% of the time (see figure 5.4), taking around five thousand clock cycles. While this reliability is not ideal, the time to produce a key is significantly lower than comparable solutions, which take around fifty thousand cycles, meaning there is room to increase the voting window for more reliability. However, the gains in reliability seem to diminish as the voting window increases, making it unclear what kind of numbers can be obtained by this technique. Memory cell classification yielded worse reliability and performance than TMV in the current configuration. If the SRAM were swapped for three-port memory, the required clock cycles would be lower than that of TMV. Increasing the voting window above 255 would also significantly reduce the performance of TMV, as the counter size would have to be increased to 16-bit.

Whether it is feasible to eliminate error correction and enrollment in SRAM PUFs using this technique comes down to the reliability required and how much time is available to reconstruct the key. A limiting factor of the experiments conducted is that all available data is taken at a single temperature, making it hard to say whether the results represent in-field reliability. How chip aging will affect reliability has not been addressed in this work.

Future work on this subject should include a study on privacy amplification techniques that can make consistently stronger keys. Since this requires extracting a larger number of bits from the PUF to be compressed into a key, it will likely negatively impact reliability and performance. Performing tests in a larger set of conditions will also be important in qualifying the PUF for use in embedded applications. This will include extracting data from the memory at a broader range of voltages and temperatures. Lastly, it could be interesting to investigate compliance with standards on the subject, such as ISO/IEC 20987 and NIST SP 800-90.

Bibliography

[1] Charles Herder, Meng-Day Yu, Farinaz Koushanfar, and Srinivas Devadas. «Physical Unclonable Functions and Applications: A Tutorial». In: *Proceedings of the IEEE* 102.8 (2014), pp. 1126–1141. DOI: 10.1109/JPROC.2014.2320516.

[2] S. Ravi, A. Raghunathan, and S. Chakradhar. «Tamper resistance mechanisms for secure embedded systems». In: *17th International Conference on VLSI Design. Proceedings.* 2004, pp. 605–611. DOI: 10.1109/ICVD.2004.1260985.

[3] Amir Ali Pour, Vincent Beroulle, Bertrand Cambou, Jean-Luc Danger, Giorgio Di Natale, David Hely, Sylvain Guilley, and Naghmeh Karimi. «PUF Enrollment and Life Cycle Management: Solutions and Perspectives for the Test Community». In: *2020 IEEE European Test Symposium (ETS).* 2020, pp. 1–10. DOI: 10.1109/ETS48528.2020.9131578.

[4] Georgios Selimis, Mario Konijnenburg, Maryam Ashouei, Jos Huisken, Harmke de Groot, Vincent van der Leest, Geert-Jan Schrijen, Marten van Hulst, and Pim Tuyls. «Evaluation of 90nm 6T-SRAM as Physical Unclonable Function for secure key generation in wireless sensor nodes». In: *2011 IEEE International Symposium of Circuits and Systems (ISCAS).* 2011, pp. 567–570. DOI: 10.1109/ISCAS.2011.5937628.

[5] Ravikanth Srinivasa Pappu. «Physical One-way functions». PhD thesis. Massachusetts Institute of Technology, 2001. URL: https://cba.mit.edu/docs/theses/01.03.pappuphd.powf.pdf.

[6] Blaise Gassend, Dwaine Clarke, Marten van Dijk, and Sahana Devadas. «Silicon physical random functions». In: *Proceedings of the ACM Conference on Computer and Communications Security.* 2002, pp. 148–160. DOI: 10.1145/586110.586132.

[7] Jorge Guajardo, Sandeep S. Kumar, Geert-Jan Schrijen, and Pim Tuyls. «FPGA Intrinsic PUFs and Their Use for IP Protection». In: *Cryptographic*

Hardware and Embedded Systems - CHES 2007. 2007, pp. 63–80. DOI: 10.1007/978-3-540-74735-2_5.

[8] Michael Vai, David J Whelihan, Benjamin R Nahill, Daniil M Utin, Sean R O'Melia, and Roger I Khazan. «Secure embedded systems». In: *Lincoln Laboratory Journal* 22.1 (2016), pp. 110–122. URL: https://www.ll.mit.edu/sites/default/files/publication/doc/secure-embedded-systems-vai-108569.pdf.

[9] Daniel E. Holcomb, Wayne P. Burleson, and Kevin Fu. «Power-Up SRAM State as an Identifying Fingerprint and Source of True Random Numbers». In: *IEEE Transactions on Computers* 58.9 (2009), pp. 1198–1210. DOI: 10.1109/TC.2008.212.

[10] Florian Wilde, Berndt M. Gammel, and Michael Pehl. «Spatial Correlation Analysis on Physical Unclonable Functions». In: *IEEE Transactions on Information Forensics and Security* 13.6 (2018), pp. 1468–1480. DOI: 10.1109/TIFS.2018.2791341.

[11] Abhranil Maiti, Vikash Gunreddy, and Patrick Schaumont. «A Systematic Method to Evaluate and Compare the Performance of Physical Unclonable Functions». In: *IACR Cryptology ePrint Archive* 2011 (Jan. 2011), p. 657. DOI: 10.1007/978-1-4614-1362-2_11.

[12] E.R. Berlekamp. «The technology of error-correcting codes». In: *Proceedings of the IEEE* 68.5 (1980), pp. 564–593. DOI: 10.1109/PROC.1980.11696.

[13] Balaji Narasimham, Dan Reed, Saket Gupta, Ennis T. Ogawa, Yifei Zhang, and J. K. Wang. «SRAM PUF quality and reliability comparison for 28 nm planar vs. 16 nm FinFET CMOS processes». In: *2017 IEEE International Reliability Physics Symposium (IRPS)*. 2017, PM-11.1-PM–11.4. DOI: 10.1109/IRPS.2017.7936393.

[14] Lawrence T. Clark, Sai Bharadwaj Medapuram, Divya Kiran Kadiyala, and John Brunhaver. «Physically Unclonable Functions Using Foundry SRAM Cells». In: *IEEE Transactions on Circuits and Systems I: Regular Papers* 66.3 (2019), pp. 955–966. DOI: 10.1109/TCSI.2018.2873777.

[15] Shin-ichi O'uchi et al. «Robust and compact key generator using physically unclonable function based on logic-transistor-compatible poly-crystalline-Si channel FinFET technology». In: *2015 IEEE International Electron Devices Meeting (IEDM)*. 2015, pp. 25.6.1–25.6.4. DOI: 10.1109/IEDM.2015.7409767.

[16] Mario Barbareschi, Ermanno Battista, Antonino Mazzeo, and Nicola Mazzocca. «Testing 90 nm microcontroller SRAM PUF quality». In: *2015 10th International Conference on Design & Technology of Integrated Systems in Nanoscale Era (DTIS)*. 2015, pp. 1–6. DOI: 10.1109/DTIS.2015.7127360.

[17] Mudit Bhargava, Cagla Cakir, and Ken Mai. «Reliability enhancement of bistable PUFs in 65nm bulk CMOS». In: *2012 IEEE International Symposium on Hardware-Oriented Security and Trust*. 2012, pp. 25–30. DOI: 10.1109/HST.2012.6224314.

[18] Roel Maes and Vincent van der Leest. «Countering the effects of silicon aging on SRAM PUFs». In: *2014 IEEE International Symposium on Hardware-Oriented Security and Trust (HOST)*. 2014, pp. 148–153. DOI: 10.1109/HST.2014.6855586.

[19] Yevgeniy Dodis, Leonid Reyzin, and Adam Smith. «Fuzzy Extractors: How to Generate Strong Keys from Biometrics and Other Noisy Data». In: *Advances in Cryptology - EUROCRYPT 2004*. 2004, pp. 523–540. ISBN: 978-3-540-24676-3. DOI: 10.1007/978-3-540-24676-3_31.

[20] Sergei Skorobogatov. *Low temperature data remanence in static RAM*. Tech. rep. UCAM-CL-TR-536. University of Cambridge, Computer Laboratory, 2002. DOI: 10.48456/tr-536.

[21] Srivatsan Chellappa and Lawrence T. Clark. «SRAM-Based Unique Chip Identifier Techniques». In: *IEEE Transactions on Very Large Scale Integration (VLSI) Systems* 24.4 (2016), pp. 1213–1222. DOI: 10.1109/TVLSI.2015.2445751.

[22] Sanu K. Mathew et al. «16.2 A 0.19pJ/b PVT-variation-tolerant hybrid physically unclonable function circuit for 100% stable secure key generation in 22nm CMOS». In: *2014 IEEE International Solid-State Circuits Conference Digest of Technical Papers (ISSCC)*. 2014, pp. 278–279. DOI: 10.1109/ISSCC.2014.6757433.

[23] J.-P. Noel et al. «A Near-Instantaneous and Non-Invasive Erasure Design Technique to Protect Sensitive Data Stored in Secure SRAMs». In: *ESSCIRC 2021 - IEEE 47th European Solid State Circuits Conference (ESSCIRC)*. 2021, pp. 455–458. DOI: 10.1109/ESSCIRC53450.2021.9567885.

[24] Synopsys Inc. *SRAM PUF: The Secure Silicon Fingerprint*. White paper. June 2022. URL: www.synopsys.com/dw/doc.php/wp/sram-puf-secure-silicon-fingerprint-wp.pdf.

[25] *Zynq UltraScale+ Device Technical Reference Manual*. ID: UG1085 (v2.4). Advanced Micro Devices Inc. Dec. 2023. URL: docs.amd.com/r/en-US/ug1085-zynq-ultrascale-trm.

[26] Maxim Integrated Inc. *How ChipDNA Physically Unclonable Function Technology Protects Embedded Systems*. White paper. Application Note 6767. 2014. URL: https://pdfserv.maximintegrated.com/en/an/ChipDNA-Unclonable-Protects-Embedded-Systems.pdf.

[27] eMemory Technology Inc. *Hardware Security with NeoPUF Solutions*. White paper. June 2018. URL: www.ememory.com.tw/Content/Upload/files/attachment/20180130125423186117024_en.pdf.

[28] Siemens EDA. *QuestaSim*. Fact Sheet. Document reference: 85329-D5 5/23 K. 2023. URL: https://static.sw.cdn.siemens.com/siemens-disw-assets/public/QzFgMxW5gizEDRIAZYTQE/en-US/Siemens-SW-QuestaSim-FS-85329-D5.pdf.

[29] Maha Kooli, Antoine Heraud, Henri-Pierre Charles, Bastien Giraud, Roman Gauchi, Mona Ezzadeen, Kévin Mambu, Valentin Egloff, and Jean-Philippe Noel. «Towards a Truly Integrated Vector Processing Unit for Memory-bound Applications Based on a Cost-competitive Computational SRAM Design Solution». In: *ACM Journal on Emerging Technologies in Computing Systems* 18 (2022), pp. 1–26. DOI: 10.1145/3485823.

[30] *ModelSim Command Reference Manual*. Software Version 10.5c. Mentor Graphics Corporation. 2016. URL: https://ww1.microchip.com/downloads/aemdocuments/documents/fpga/ProductDocuments/ReleaseNotes/modelsim_me_v105c_ref.pdf.

[31] Iluminada Baturone, Miguel A. Prada-Delgado, and Susana Eiroa. «Improved Generation of Identifiers, Secret Keys, and Random Numbers From SRAMs». In: *IEEE Transactions on Information Forensics and Security* 10.12 (2015), pp. 2653–2668. DOI: 10.1109/TIFS.2015.2471279.

[32] Roel Maes, Anthony Van Herrewege, and Ingrid Verbauwhede. «PUFKY: A Fully Functional PUF-Based Cryptographic Key Generator». In: *Cryptographic Hardware and Embedded Systems – CHES 2012*. 2012, pp. 302–319. ISBN: 978-3-642-33027-8. DOI: 10.1007/978-3-642-33027-8_18.

[33] Intrinsic ID B.V. *QuiddiKey 100 Hardware IP*. Product Brief. 2023. URL: https://web.archive.org/web/20230801042458/https://www.intrinsic-id.com/wp-content/uploads/2023/07/QuiddiKey-100-Hardware-IP-v1-0-0-Product-Brief.pdf.

Appendix A

CyberPUF Function Descriptions

bits2Bytes

```
byteCompressedElement = bits2Bytes(element)
```

Converts a bit matrix (one cell per index) to a byte matrix (8 cells per index). Effectively reduces the number of columns by a factor of 8. If the number of columns in the input is not a multiple of 8 then padding with zeroes will be applied on the right side of the matrix.

byte2Bits

```
bitExpandedElement = byte2Bits(element)
```

Converts a byte matrix into a bit matrix, expanding the number of columns by 8.

calculatepMatrix

```
pMatrix = calculatepMatrix(restartBitmaps, delay, figNr)
```

Finds the average of each cell based on a set of bit matrices. Optional delay argument can be used for timing accurate simulation, while figNr is used for plotting the resulting P-matrix.

calcPDistribution

```
[vxDecale, ivyN, sigma, discretey, alpha, beta, gamma] =
   calcPDistribution(pMatrix, nBins, figNr, nRestarts,
   dlabel)
```

Returns the p-distribution properties (see createMixedPDistribution of a given p-matrix. Takes as input and number of bins. Optional arguments are used for plotting.

createMixedPDistribution

```
[vxDecale,composite,discretey,cf] =
   createMixedPDistribution(nBins, alpha, DELTA, deltap,
   sigmaP, figNr)
```

Creates a probability distribution with discrete (p=0 or p=1) and continuous (0<p<1) parts. Alpha is the proportion of bits in the discrete part and delta delta is the asymmetry in the discrete part. The proportion of bits with p=0 will be $(alpha + delta)/2$ and the proportion of bits with p=1 will be $(alpha - delta)/2$. DeltaP is the proportion of bits which are part of the decaying "bumps" at the edges of the distribution, which has a spread of sigmaP. The remainder of the bits $(1 - alpha - deltaP)$ will be uniformly distributed in the continuous part. Note: if sigmaP is 0, then deltaP is set to 0

displayBitmap

```
displayBitmap(figNr, TMV, mode)
```

Displays a given bitmap. Three available modes: 0-Black and white. 1-Grayscale, 2-White/Gray/Red.

dumpAISTPufPerf

```
dumpAISTPufPerf(LenID, NumID, BitIteration, pufResult,
   fout2, verb)
```

Dumps AIST PUF performance data (see getAISTPufPerf) to a file.

findTMV

```
TMV = findTMV(pMatrix, figNr)
```

Rounds a given p-matrix to obtain a Temporal Majority Vote bitmap. Optional figNr argument to plot.

fractionalHD

```
delta = fractionalHD(bitmap, refBitmap)
```

Returns the fractional Hamming Distance between two bitmaps.

getAISTPufPerf

```
[pufResult, retArray] = getAISTPufPerf (storeAllPUF, verb)
```

Returns the quality results of PUF data as described by AIST. The input is a three-dimensional matrix with dimensions $[trials, keys, bits]$.

getPmatrixFromMixedPDistr

```
combA = getPMatrixFromMixedPDistr(M, N, vxDecale,
    continuous, discretey, verb)
```

Creates a M by N probability matrix based on a mixed P-distribution. Takes as arguments the rows and columns of the output matrix, and the properties of the target distribution (see *createMixedPDistribution*).

hexFile2mem

```
matrix = hexFile2mem(path)
```

Reads the target .hex file into memory as a bit matrix.

initCyberPUF

```
initCuberPUF
```

Script to initialize the CyberPUF package. NOTE: clears all variables in the workspace and closes all figures when executed.

mapBitmap2PhysicalArray

```
ret = mapBitmap2PhysicalArray(bitmap, muxf)
```

Rearranges a bitmap from $[rows, cols]$ to $[rows/muxf, cols \cdot muxf]$. Can be used to reconstruct data read from a bus which is narrower than the array.

printPUFInfo

```
[nbOfKeys, keyLength, nTrials] = printPUFInfo(storeAllPUF,
     verb)
```

Returns the number of keys, word length, and number of trials in a dataset arranged as $[trials, keys, bits]$.

selectOneBitmap

```
ret = selectOneBitmap(array,index)
```

Returns the bitmap at index from array.

simulateMatrices

```
[combA, vxCheck2, vyCheck2, discyCheck2, alphaCheck2,
   uLevelCheck2, deltaPCheck2] = simulateMatrices( pMatrix
   , nRestarts, nBins, figNr)
```

Simulates SRAM initializations based on a probability matrix. Takes as arguments the target P-matrix and the number of restarts to simulate. Optional arguments are used to plot the distribution. Returns a set of bit matrices and the properties of the resulting distribution (see *createMixedPDistribution*).

writeHex

```
status = writeHex (fname, B, Baddr, verb, headerFilePath)
```

Exports a **byte** matrix (see *bits2Bytes*) as a .hex file. Takes as argument a path including the file name for the output, and a byte matrix. Optional argument include a list of addresses to output (default is 0...Nrows-1), verbosity, and a path to a header to include in the output file. Loading the hex file in Questasim requires a header specifying the format, addressradix, dataradix, version, and wordsperline. Returns the status of the operation (0 if successful, else -1).